Desarrollo de un Sistema Multiagente Basado en Creencias, Deseos e Intenciones para Modelar Personajes Autónomos en Videojuegos utilizando Jason y Unity

Sergio González Jiménez

FACULTAD DE INFORMÁTICA
DEPARTAMENTO DE INGENIERÍA DEL SOFTWARE
E INTELIGENCIA ARTIFICIAL
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Grado en Ingeniería Informática

Madrid, 8 de junio de 2018

Director: Prof. Dr. Federico Peinado Gil
Codirector: Javier Vela Ramos

Agradecimientos

A Federico Peinado y Javier Vela por su apoyo a lo largo del proyecto, facilitándome la labor de realizar en solitario este trabajo final. También a toda la gente de Narratech que colaboró en las pruebas y en las revisiones de la memoria y aportó su feedback.

También a todos los amigos que han puesto de su parte para que pudiera dedicarle más tiempo al trabajo. Por último, y especialmente, a Sergio. Sin su ayuda y experiencia en Unity jamás podría haber avanzado tanto como lo he hecho. Su apoyo ha sido fundamental en este trabajo.

Índice general

Resumen

Los videojuegos siempre han querido incorporar personajes autónomos como parte de sus mundos de ficción. El problema de la IA en personajes de videojuego no es tanto el ser avanzada sino que requiere credibilidad, y ser suficientemente expresiva como para que el jugador "entienda" lo que hace el personaje. Un modelo interesante de IA es el que considera lo que el personaje cree saber sobre el mundo, los deseos que tiene y las intenciones que es capaz de desarrollar para satisfacerlos en forma de planes.

Usar técnicas simbólicas de IA y herramientas pensadas para la depuración de sistemas multiagente con este tipo de arquitecturas puede ayudar a desarrollar y convencer a la industria de que es posible tener personajes inteligentes y capaces de "explicar" sus comportamientos.

Con el fin de explorar nuevas tecnologías para crear personajes en videojuegos, se propone utilizar el modelo de Creencias-Deseos-Intenciones y una combinación de plataformas (Unity como entorno de desarrollo de videojuegos, y Jason como plataformas para desarrollar y depurar sistemas multiagente BDI) para conseguir personajes inteligentes.

Se propone para ello un conector *UniJason*, basado en otro TFG (Sánchez-López, Romero, y Martín-Solís, 2016), con un protocolo mejor definido de mensajes que se intercambian de manera bidireccional, y un diseño genérico aplicable a cualquier tipo de videojuego.

Para probar este sistema se ha desarrollado un videojuego, *Miners of the Broken Planet* (en homenaje a la superproducción española de Mercurysteam, *Raiders of the Broken Planet*). En éste, el jugador puede manejar a distintos personajes, cada uno con un rol, tareas y personalidad diferentes.

Las pruebas con usuarios reales realizadas confirman que el conector funciona, y que el juego también es comprensible para las personas que lo juegan.

Palabras clave

Desarrollo de Software, Desarrollo de Videojuegos, Informática del Entretenimiento, Inteligencia Artificial, Sistemas Multiagente, Ingeniería del Software, Jugabilidad Emergente

DEVELOPMENT OF A MULTI-AGENT SYSTEM BASED ON BELIEFS, DESIRES AND INTENTIONS FOR MODELLING AUTONOMOUS CHARACTERS IN VIDEO GAMES USING JASON AND UNITY

Abstract

Video games have always wanted to incorporate autonomous characters as part of their fiction worlds. The problem with AI in video game characters is not being advanced, but to be credible, and to be sufficiently expressive for the player to 'understand' what the character does. An interesting model of AI is the one that considers what the character thinks he knows about the world, the desires he has in that world and the intentions he is able to develop to satisfy them in the form of plans.

Using symbolic AI techniques and tools designed for debugging multi-agent systems with this type of architecture can help develop and convince the industry that it is possible to have intelligent characters capable of 'explaining' their behaviour.

In order to explore new technologies for creating characters in videogames, it is proposed to use the Beliefs-Wishes-Intentions model and a combination of platforms (Unity as a videogame development environment, and Jason as platforms for developing and debugging multi-agent BDI systems) to achieve intelligent characters.

A UniJason connector based on another TFG is proposed for this purpose. A one with a better defined protocol of messages that are exchanged bidirectionally, and a generic design applicable to any type of videogame.

To test this system, a video game has been developed, *Miners of the Broken Planet* (an homage to Mercurysteam's Spanish superproduction, Raiders of the Broken Planet). In this game, the player can manage different characters, each with a different role, tasks and personality.

Testing with real users confirms that the connector works, and that the game is understandable to the people who play it.

Keywords

Software Development, Video game Development, Entertainment Informatics, Artificial Intelligence, Multi-agent Systems, Software Engineering, Emerging Gameplay

Capítulo 1

Introducción

Las aplicaciones de la Inteligencia Artificial (IA) al Desarrollo de Videojuegos es un tema de gran interés, tanto para los expertos como para los consumidores. En los últimos años, esta sinergia ha propiciado la aparición de herramientas para tratar de reducir el número de recursos y la gran cantidad de esfuerzo que toda gran producción presupone.

Mientras tanto, el progreso tecnológico y el acceso, por parte de los consumidores, a *hardware* más avanzado a precios asequibles ha facilitado el desarrollo de videojuegos que ofrecen extensos mundos que explorar, habitados por seres que interactúan con el jugador y con su entorno.

También se ha hecho más asequible, de cara a las empresas, el competir en la industria del videojuego, ofertando funcionalidades e innovaciones tecnológicas a niveles nunca vistos en su corta historia. Ésto, empuja a los desarrolladores a inventar nuevas fórmulas para destacar en un medio con un público cada vez más exigente, ya sea para abrirse camino en un mercado independiente saturado con juegos de todos los estilos imaginables; o para mantener una posición de dominio por parte de las grandes empresas, renovando la forma en que construyen las nuevas entregas de sus franquicias de éxito.

Sin embargo, a pesar de estos avances, la Inteligencia Artificial aplicada a la jugabilidad no ha mejorado tanto en los últimos años. Videojuegos como *Fallout 4* (Bethesda Softworks, 2015) manifiestan algunas carencias presentes en títulos anteriores, como *TES: Skyrim* (Bethesda Softworks, 2011), —que a su vez las hereda de *TES: Oblivion* (Bethesda Softworks, 2006)— en lo que se refiere a la IA de sus *NPCs*(*Non Playable Characters*). Casi una década de diferencia entre el más antiguo y el más moderno.

Este estancamiento puede deberse a varios factores relacionados. El primero de ellos es una cuestión de presupuesto; el segundo, la gran dificultad que supone publicitar las mejoras que la Inteligencia Artificial ofrece en un videojuego, en lugar de desarrollar un apartado gráfico que hable por sí solo; y por último, lo sutil que puede resultar, a efectos prácticos, un esfuerzo que supone muchos recursos.

Al margen de la IA, en los últimos años también destaca el creciente interés por la jugabilidad emergente. Ésta se define como el conjunto de dinámicas no prefabricadas que surge a partir de las interacciones del jugador dentro del juego. Es decir, los usos que el jugador puede dar a las diferentes herramientas que contiene el juego, para resolver problemas de maneras no diseñadas explícitamente en la creación del videojuego.

En la búsqueda de una forma de dotar de sentido al uso de IA avanzada en el desarrollo del videojuego, de una manera asequible y relativamente sencilla, se decide optar por un paradigma que represente el comportamiento de los personajes de un videojuego de forma no guionizada. También se intenta evitar la complejidad y el acoplamiento entre tareas que habría en estructuras como los árboles de comportamiento (*Behavior Trees*[1] de Unreal Engine).

[1]Behavior Trees (Unreal Engine): `https://docs.unrealengine.com/en-us/Engine/AI/BehaviorTrees`

Es por ello que para representar el conocimiento de los agentes, se ha escogido el modelo cognitivo *BDI* (*Beliefs, Desires, Intentions*). Éste se caracteriza por proponer que los modelos mentales que los seres humanos tenemos sobre nuestro entorno son teorías acerca del mismo. Esas teorías representan las Creencias, que pueden ser o no verdaderas, que dan contexto a cualquier posible plan a desarrollar, las Intenciones, las cuales suponen los medios para lograr un fin, y los Deseos del agente.

1.1. Propósito del trabajo

Teniendo en cuenta estas premisas, se desarrolla este Trabajo de Fin de Grado, cuyo propósito es explorar y justificar el uso de nuevos enfoques para aplicar Inteligencia Artificial a videojuegos. En concreto, aplicando la arquitectura cognitiva *BDI* al comportamiento de personajes no jugables. También, como cualquier TFG, tiene como fin aplicar varios de los conocimientos adquiridos a lo largo del grado, y profundizar en campos y herramientas nuevas.

Para lograr estos propósitos, se han utilizado principalmente dos herramientas que son importantes en sus campos: el intérprete basado en Java, Jason[2], para el desarrollo de la IA de los personajes del videojuego, apoyada en un Sistema Multiagente; y Unity[3], el entorno de desarrollo de videojuegos de referencia en la industria.

Este proyecto consta de dos partes diferenciadas:

La primera, el desarrollo de una herramienta genérica y extensible que comunica el videojuego, implementado desde el motor Unity, con la inteligencia artificial de los personajes del mismo, desarrollada en Java y utilizando Jason.

[2]Jason Interpreter: http://jason.sourceforge.net/
[3]Unity Engine: https://unity3d.com

La segunda, el diseño y desarrollo de un videojuego que aprovecha la tecnología que proporciona Jason, con personajes que actuan de maneras distintas en función de lo que saben de sí mismos y del entorno. Teniendo en cuenta la complejidad de esta tarea, se acudió al apoyo del codirector del trabajo, Javier Vela Ramos, quien trabaja como *Game Designer* en *Mercury Steam*[4]. Con su ayuda se elaboró un diseño que permitó demostrar algunas de las bondades del *BDI* en el videojuego.

1.2. Estructura del trabajo

Una vez explicadas el propósito de este trabajo, y el contexto en que se desarrolla, a continuación se describe el contenido de los siguientes capítulos presentes en la memoria. El Capítulo 2 es el estado de la técnica. Revisa, por orden, las aplicaciones de la IA en videojuegos concretos, a nivel teórico; herramientas reales, que ya existen para Unity, que desarrollan la IA; y por último otros proyectos y estudios similares al realizado. En el Capítulo 3 se detallan los objetivos de este proyecto, especificando los requisitos que sirven de base para todo el desarrollo posterior. El plan de trabajo general, la metodología y las herramientas relacionadas con la comunicación y la organización de trabajo que han sido utilizadas se presentan en el Capítulo 4.

El Capítulo 5 expone el grueso de las actividades estructurales del proceso de desarrollo software de este trabajo. En él se detallan análisis, diseño e implementación del conector genérico entre Jason y Unity, que denominamos *UniJason*; y del prototipo de videojuego, *Miners of the Broken Planet*, formalizado en un documento de diseño de videojuego.

Las pruebas realizadas sobre el prototipo del videojuego y los resultados obtenidos se discuten en el Capítulo 6. Finalmente, el Capítulo 7 recoge las conclusiones sobre este proyecto, las cuales han generado varias líneas de trabajo futuro interesantes para seguir mejorando tanto el conector como el prototipo de videojuego desarrollado.

[4]Mercury Steam: `https://www.mercurysteam.com/`

Capítulo 2

Estado de la técnica

En este capítulo se revisan algunas aplicaciones de la IA en videojuegos profesionales donde se enmarca este trabajo, así como herramientas disponibles en Unity para crear este tipo de IAs. Finalmente se consideran también proyectos de carácter investigador que apuntan en posibles direcciones por donde es posible mejorar el comportamiento de los personajes en videojuegos.

2.1. Inteligencia Artificial en Videojuegos profesionales

En este apartado se exponen algunos ejemplos de aplicaciones de IA en videojuegos profesionales, con la intención de hacer una imagen aproximada del estado de ésta en el escenario de las grandes producciones.

El motivo de revisar las aplicaciones de la IA en videojuegos profesionales es poder evaluar qué sistemas incluyen y contemplar su margen de mejora. Los siguientes casos de estudio están ordenados cronológicamente, del más antiguo al más moderno.

2.1.1. *Killzone 3*, *bots* en multijugador

Killzone 3(Guerrilla Games, 2011) incluyó un sistema jerarquizado de IA en su vertiente multijugador. Dicha vertiente posee un modo de juego, *Warzone*, que enfrenta hasta a 12 jugadores en dos equipos, los *ISA* y los *Helgast*, en siete modos distintos de juego en el mismo mapa. Para facilitarle al jugador el aprendizaje de estos modos de juego multijugador, se añadieron *bots* que imitaban el comportamiento de jugadores experimentados, sabiendo aprovechar las características y habilidades de la clase que representasen, aliviando algo de carga a los jugadores menos experimentados.

Figura 2.1: Escuadrón de soldados *Helgast*

Este sistema jerárquico se divide en tres capas: la de estrategia, la de escuadrones, y la de los *bot* individuales. La primera contiene la IA del comandante, que monitoriza el estado del juego, asigna bots a cada escuadrón y propone objetivos. La segunda contiene las IA de cada escuadrón. Éstas se encargan de traducir los objetivos del comandante en ordenes para los bots, coordinar sus movimientos y comprobar el cumplimiento de los objetivos.

Finalmente, la IA de cada *bot* se caracteriza por seguir las órdenes del escuadrón, pero dispone de cierta libertad sobre cómo ejecutarlas. En cualquier caso, la IA del *bot* se ocupa de gestionar el combate y uso de habilidades de su clase. De esta manera, los *bots* pueden elegir a quién atacar, desde qué posición y con qué arma, e incluso ignorar órdenes del escuadrón en pos de su supervivencia.

2.1.2. *The Last of Us*, conductas del compañero

The Last of Us (Naughty Dog, 2013) es un juego cuyo modo "Historia" pone al jugador en la piel de Joel, un superviviente de mediana edad que acompaña durante la mayoría del juego a Ellie, una adolescente, en un viaje a través de Estados Unidos. Van en busca de un grupo secreto, Los Luciérnagas, que pueden crear una cura para la infección que asola al planeta entero. Para sus creadores era importante crear un vínculo afectivo entre el jugador y Ellie, evitando hacer del juego una misión de escolta y evitando crear un personaje que resulte ajeno a su entorno.

Figura 2.2: Los dos protagonistas, Joel y Ellie

La IA de Ellie pretende representar el carácter curioso de una niña, que ha nacido en un mundo destruido y crecido dentro de una zona militarizada, al margen del exterior, que ahora descubre un mundo lleno de paisajes, ciudades y criaturas que jamás había visto. De esta manera, narrativa e IA se dan la mano para crear un personaje verosímil, que se siente vivo.

La base de su IA es el sistema de seguimiento. El primero calcula los movimientos del compañero en tres pasos. Primero genera un conjunto de posiciones detrás del jugador a las que el compañero se puede mover, al que se denomina *follow region*, validando aquellas posiciones que se encuentren en una trayectoria despejada (sin obstáculos) con el jugador; después, a partir de cada posición válida se traza una trayectoria en la dirección en que se mueve el jugador, generando otro conjunto de posiciones y validando sólo aquellas que no sitúen al acompañante frente a un obstáculo; y finalmente, se trazan una línea entre la posición del jugador y cada posición validada en el anterior conjunto, comprobando que no hay ningún obstáculo entre ambas posiciones, o lo que es lo mismo, que ningún obstáculo corta la linea entre ambos puntos.

Además de este sistema, también cuenta con uno de exploración que permite fijar puntos de interés con los que Ellie puede interactuar, dando al jugador la sensación de que está viva. Esto es de especial interés para el este proyecto, ya que hace uso de la IA para hacer más inmersiva la experiencia. Está relacionado con el diseño de los comportamientos de Ellie según su personalidad, aunque de una forma más "artesanal" de la buscada con este trabajo.

Sin embargo, los desarrolladores confiesan haber hecho concesiones al jugador, modificando la IA enemiga para no lastrar la experiencia de juego, haciendo a la acompañante invisible ante los enemigos. La motivación detrás de ésto es que, en ocasiones, la IA de Ellie podría hacer que se moviera a la vista de los enemigos, lo cual es un comportamiento poco coherente, revelando la posición de los protagonistas y echando a perder posibles estrategias que el jugador pretenda llevar a cabo. De este modo, conscientes de este problema en la IA del personaje, optaron por ocultarla de la percepción enemiga.

2.1.3. *Dragon Age Inquisition*, evaluación de habilidades

Dragon Age Inquisition (Bioware, 2014) es un videojuego de rol occidental que da un gran peso a las decisiones del jugador, que condicionan el desarrollo de la historia, y también al combate. El jugador va acompañado de un grupo de aliados controlados por la CPU, que le ayudan a vencer haciendo uso de sus habilidades.

Figura 2.3: El jugador enfrentándose a enemigos

Todo personaje en el juego, incluido el jugador, posee un compendio de habilidades, las cuales tienen un coste asociado en recursos limitados, como son el *maná* para los hechizos o la energía para los ataques físicos, y que pueden ser más o menos útiles dependiendo del contexto de la situación. De esta manera, la IA de los aliados necesita saber evaluar qué habilidad es más interesante realizar en cada momento, atendiendo a su coste y su utilidad.

Para ello se utiliza el *Behavior Decision System* (BDS), un sistema de puntuaciones que sirve para analizar y comparar entre sí las habilidades realizables por un personaje en un determinado momento. Debido a que algunas habilidades no tienen siempre un sólo uso, se utilizan unas estructuras, los *behavior snippets* (algo así como "fragmentos de comportamiento").

Éstos fragmentos contienen la información relativa a la forma de utilizar una habilidad de una forma concreta. Cada fragmento posee un árbol que, a medida que es recorrido, suma el valor de sus nodos dependiendo del contexto del combate. Por ejemplo, para el uso de pociones, en un nodo se podría consultar si el personaje está a la mitad de su salud, añadiéndose a la puntuación del recorrido el valor de ese nodo si es el caso, y continuando el recorrido del árbol hasta su final, devolviendo como puntuación de ese fragmento el valor total de los nodos validados.

Es responsabilidad del diseñador construir los árboles de evaluación para cada fragmento, asegurándose de que las puntuaciones que proporciona son acordes a los intervalos fijados según el ámbito de la acción del fragmento: básica, ofensiva, de apoyo y de reacción.

Pero no todas las habilidades afectan de la misma forma a todos los enemigos. Concluir que lanzar una bola de fuego es la habilidad más recomendable, tanto para enemigos que son débiles al fuego como para los que son inmunes a él, hace que la evaluación no sea óptima. De esta forma, la evaluación del fragmento no sólo se vincula a una habilidad con un determinado propósito, sino también a cada objetivo posible. Así, se evalúa para cada fragmento a cada posible objetivo, almacenando para el fragmento la puntuación del recorrido en el árbol y el objetivo que la maximiza. Una vez evaluados todos, se elige el fragmento con mayor puntuación, y se realiza esa acción.

2.1.4. *Final Fantasy XV*, coordinación de compañeros

Final Fantasy XV (Square Enix, 2015) es un título de acción RPG de mundo abierto, en el cual el príncipe Noctis, el que maneja el jugador, y sus amigos deben derrotar al imperio de Niflheim y vengar la caida de su natal Insomnia. Como su pertenencia al género Action-RPG insinúa, la presencia de combates a lo largo del videojuego es enorme, siendo un pilar principal de la experiencia. En estos combates, al igual que en Dragon Age Inquisition (Bioware, 2014), los aliados del jugador están controlados por la CPU, y crear un comportamiento satisfactorio es crucial para la experiencia de usuario.

Figura 2.4: Noctis e Ignis rodeados por enemigos

Al contrario que en el resto de casos expuestos, en éste no se trató de resolver un problema concreto, sino que se propuso y construyó una herramienta para lograr un comportamiento inteligente de los aliados. La propuesta de Final Fantasy XV (Square Enix, 2015) se llama *AI Graph*, que combina la secuencialidad de acciones de los árboles de comportamiento y la estabilidad de las máquinas de estados finitos, consiguiendo la flexibilidad que ofrece una y el control estricto de la otra, al mismo tiempo que ofrece escalabilidad en el desarrollo del sistema de toma de decisiones de cada personaje. La herramienta permite intercalar ambas estructuras en el desarrollo de cada inteligencia particular.

Como característica principal de la herramienta está la posibilidad de implementar máquinas de estados que permiten tener varios estados activos en paralelo para, por ejemplo, examinar el entorno en busca de su nuevo objetivo al mismo tiempo que seguimos atacando a un enemigo. Adicionalmente, un árbol de comportamiento puede ejecutar dos procesos simultáneos a través de un nodo paralelo, ofreciendo resultados similares.

Aunque tal vez la parte más importante del *AI Graph* es el llamado *meta-AI* o *AI Director*. Monitoriza la situación de las batallas y el comportamiento de cada personaje en ellas. Por ejemplo, cuando uno de los compañeros del equipo, sea o no el jugador, está en peligro, el director de IA elige al aliado más próximo, ordenándole que vaya en su ayuda. La toma de decisiones de los compañeros depende del *AI Graph*, pero cuando recibe una orden del director, para la ejecución de su grafo y obedece la orden.

El director puede dar cuatro tipos de órdenes: Salvar a un compañero en peligro, procurar una vía de escape al jugador si está rodeado de enemigos, seguir al jugador si escapa de la batalla, y obedecer las tácticas de equipo. Gracias a esto, el *meta-AI* puede aumentar o disminuir la tensión de cada enfrentamiento.

2.2. Herramientas de IA para Unity

En este apartado se examinan algunas herramientas de IA para Unity presentes en su Asset Store, y alguna, de especial relevancia para el presente proyecto, desarrollada por alumnos de la Universidad Complutense de Madrid. El objetivo aquí es ofrecer un retrato de las tecnologías disponibles, comentando herramientas de propósitos diferentes.

2.2.1. *ANN Perceptron*

ANN Perceptron (Virtualstar, 2018) la primera —y única— herramienta catalogada dentro de la categoría *Machine Learning* de la Asset Store, aunque en realidad no sea la única presente en toda la tienda. Lo que ofrece esta herramienta es generar, de forma transparente al desarrollador, redes neuronales de dos tipos: De *Back Propagation* y de Random Generation.

La herramienta permite ajustar las características de la red neuronal. Entre éstas se incluyen cosas tan básicas como el número de entradas y salidas, el número de capas, el número de neuronas en cada capa. También permite ajustar el sesgo, la velocidad y el ratio de aprendizaje, el error máximo permitido, entre otras características más complejas. Todos estos ajustes se realizan de forma sencilla, y permite visualizar los cambios en la red en tiempo real, así como su evolución a lo largo del tiempo.

2.2.2. *Discrete Bayesian Network*

Las redes bayesianas tienen especial utilidad en IA cuando se trata con información con cierto grado de incertidumbre, como es el caso de los videojuegos, pudiendo modelizar comportamientos para los personajes presentes en un videojuego, en función de cómo se relacionan las variables a tener en cuenta para el modelo de dichos comportamientos.

Discrete Bayesian Network (Jacky Chen, 2016) permite implementar redes bayesianas para relacionar datos discretos del entorno de juego y permitir a los personajes que hacen uso de ellas tomar decisiones.

2.2.3. *Behavior Designer*

Behavior Designer (Opsive, 2014) es una herramienta utilizada para crear árboles de comportamiento. Ofrecer un editor de árboles robusto y relativamente intuitivo, permitiendo al desarrollador crear nuevas tareas que utilizar en los árboles haciendo uso de una API propia. También incluye tareas ya implementadas, e integración con software de terceras compañías para la elaboración de IAs complejas. Para facilitar su uso, la herramienta también implementa un depurador visual en tiempo real.

Entre los títulos desarrollados con esta herramienta se encuentran el español *Immortal Redneck* (Crema Games, 2017), un *first-person shooter rogue-like*, y *Warcube* (Haven Made, Craigz, 2017), un juego arcade en perspectiva cenital, y *Project Wight* (Outsider Studios, TBD), un RPG de corte occidental. Haciendo énfasis en los géneros de éstos títulos, sus creadores defienden el carácter polivalente de su herramienta.

2.2.4. *Isomonks*

Isomonks (Sánchez-López y cols., 2016) es un proyecto de alumnos de la UCM que consistió en la creación de un conector entre IsoUnity (Pérez-Colado y Pérez-Colado, 2014), un *plugin* para crear videojuegos con perspectiva isométrica en Unity, y Jason, el sistema multiagente basado en *AgentSpeak*.

Dicha herramienta se diseñó e implementó en el marco de una demo bastante sencilla, que consistía en un tributo a *La Abadía del Crimen* (Opera Soft, 1987) en el que unos monjes se movían por un pequeño mapa, condicionados por la activación de algunos eventos provocados al mover a un personaje —controlado por el jugador— a distintas salas, o interactuando con una campana presente en el escenario.

No puede decirse que sea una herramienta comparable a las demás propuestas en esta sección, debido a que no es ni genérica ni extensible —al estar atada a la implementación de una demostración— aunque por la relación que tiene con el presente proyecto merece la mención.

Sin embargo, es de especial relevancia en este proyecto, ya que ofrece una idea en la que se inspira el diseño e implementación del conector de este trabajo final.

2.2.5. *Emerald*

Emerald (Black Horizon Studios, 2015) es una herramienta que concentra algunas disciplinas de la IA en videojuegos: Conductas, percepción y planificación de rutas (*pathfinding*). Como intereses principales de la herramienta destacan varios puntos.

El primer punto aborda los comportamientos predefinidos, existiendo cinco diferentes para los *NPC*: *Pasivo*, *Agresivo*, *Cauteloso*, *Compañero* y *Mascota*. Adicionalmente, *Pasivo*, *Agresivo* y *Cauteloso* están condicionados por distintos temperamentos: *Cobarde*, *Valiente* y *Temerario*, variando su comportamiento dependiendo de cual sea el temperamento asignado al *NPC*.

El segundo punto incluye un sistema de visión que puede ser utilizado por la IA de los *NPCs* para atacar o huir de los objetivos que estén en su campo de visión, para evitar que reaccionen frente a otros personajes a través de las paredes.

También destaca la implementación *built-in* de la interfaz, que dispondrá en la pantalla barras de salud, nivel de los personajes presentes y otros indicadores; también la del sistema de combate a distancia, permitiendo a los *NPCs* entrar en combates a distancia, permitiendo al desarrollador personalizar el proyectil que dispara un personaje: Su sonido, tiempo, efecto de animación y otros. *Emerald* aplica automáticamente todos los *scripts* y componentes necesarios al proyectil para que funcione.

Por último, se destaca el sistema de puntos de referencia, gracias a los cuales el desarrollador puede marcar, borrar y recolocar puntos por los que el personaje transitará,sin desobedecer a los comportamientos predefinidos (Pasivo, Agresivo, ...) anteriormente comentados. De este modo, reacciona a posibles objetivos que se crucen en su camino, aunque tras acabar con su objetivo (o perderlo de vista), continuará su ruta con normalidad.

2.2.6. *Apex Utility AI*

La última herramienta es otra colección de IA que, a diferencia de las demás, ha sido empleada en títulos comerciales de alto presupuesto con IA destacable.

Apex Utility AI (Apex Game Tools, 2016) pone a disposición del desarrollador una herramienta de alto rendimiento, con un editor sencillo, que permite conectar distintas utilidades de decisión representadas a través de cajas para construir flujos de decisiones y árboles de decisión. La herramienta permite depurar la IA de los personajes en tiempo real, haciendo uso de puntos de ruptura.

Su interfaz es cómoda, permitiendo arrastrar, copiar-pegar, y deshacer- hacer con las cajas de las utilidades por la pantalla del editor.

Las utilidades de decisión se componen de puntuaciones fijas o de funciones (curvas de decisión) que evalúan la acción que es más recomendable realizar en cada momento, en función de los parámetros del entorno que intervengan en dicha función.

Por último, los creadores de *Apex Utility* AI destacan la naturaleza multipropósito de su herramienta, pudiendo crear IA para coordinar grupos, hacer razonamiento táctico, mantener diálogos dinámicos, y seleccionar metas, evaluando todas las realizables y escogiendo siempre la que tenga una mayor utilidad.

2.3. Proyectos e investigaciones de IA para videojuegos

En esta sección realizamos una revisión un tanto superficial de proyectos de investigación en IA para videojuegos. El objetivo de este trabajo no es realizar una contribución científica en este campo, pero sí se ha considerado interesante saber hacia donde apuntan estos proyectos para dirigir nuestros esfuerzos a la hora de crear nuestro software hacia las áreas más prometedoras.

2.3.1. Adaptación de misiones a los intereses del jugador

En *Planning for Individualized Experiences with Quest-Centric Game Adaptation* (Li y Riedl, 2010), sus autores parten de la premisa de que se puede mejorar la experiencia jugable a través de la adaptación a los gustos deseos y motivaciones de los jugadores. De entre todos los frentes posibles, ellos trabajan la adaptación a los intereses del jugador, de los objetivos y eventos que intervienen en una "linea de misiones" (*questline*), para juegos que giran en torno a la resolución de misiones. Su objetivo es, en otras palabras, involucrar las aspiraciones y los deseos del jugador en la coherencia narrativa.

Gracias a su sistema, aseguran poder incrementar la variedad de misiones que se ajustan a los deseos del jugador, incrementando la rejugabilidad a la vez que mantienen la coherencia narrativa entre todos los eventos involucrados en una misión, y la originalidad y creatividad de las historias creadas por los diseñadores. La intención de este sistema no es crear historias por si solas, sino modificar las que han sido hechas por diseñadores humanos.

Explican cómo una narrativa es coherente cuando todos los eventos de ésta contribuyen —tienen efecto— en el desenlace final. Sugieren que algunos eventos tienen especial relevancia y significado en una trama, otros confieren un contexto que a menudo desemboca en la aparición de las misiones, y todos ellos dan forma a lo que llaman *Core Set*. Pero la coherencia narrativa no se obtiene con estos eventos por sí mismos, sino también asegurando la ausencia de fallos en la coherencia. De entre todos los fallos que puede hacen de incoherente una narrativa, distinguen dos de especial relevancia: los *dead ends* o callejones sin salida y los esfuerzos superfluos.

Los callejones sin salida se caracterizan por no aportar de una forma significativa a la forma en que se desarrollan los eventos de la misión. En el caso de mantener la coherencia con las intenciones del jugador, un evento que le conduce a una recompensa que no aporta nada significativo a sus metas, es considerado un callejón sin salida. El ejemplo que ponen es el de un jugador que quiere ser muy rico, pero como recompensa de una misión obtiene una espada mágica, cuando ésta no contribuye de forma directa a su deseo de ser rico.

Los esfuerzos superfluos son aquellos eventos que obligan al jugador a hacer cosas para luego volver a deshacerlas, sin que en ese proceso haya habido un efecto colateral de interés para el desarrollo de la misión.

Su algoritmo consiste en descomponer los planes abstractos que dan forma a una misión en forma de acciones primitivas relacionadas con el plan abstracto. Después recorre y elimina aquellos eventos que no interesan al jugador, y los reemplaza por otros que sí lo hacen. La complicación aparece cuando al borrar algunos eventos, se pueden producir callejones sin salida, que si son sustituidos por otros eventos, corren el riesgo de conducir a esfuerzos superfluos. Para evitar esto, si el evento que es irrelevante no es parte de una descomposición de un plan, simplemente se borra sin riesgo a producir incoherencias narrativas; pero si lo es, entonces todos los demás eventos que sean parte de un plan descompuesto también son eliminados, y el plan es marcado como no-descompuesto, pudiendo ser reemplazado sin riesgos.

Según sus pruebas la eliminación de callejones sin salida y esfuerzos superfluos es útil a la hora de generar narrativas consistentes, si bien admiten que hay muchos más fallos que pueden afectar a la coherencia narrativa de una historia.

2.3.2. Sistemas multiagente en videojuegos *sandbox*

En la publicación *Multi-agent Systems and Sandbox Games* (Ocio y Burgos, 2009) se reflexiona y se sugiere una aproximación sobre el uso de sistemas multiagente en videojuegos de tipo *sandbox* —a grandes rasgos, videojuegos de mundo abierto—. En el contexto de su publicación, los primeros *sandbox* estaban ambientados en grandes ciudades que emulan en complejidad a las del mundo real, siendo *Grand Theft Auto* (Rockstar Games, 1997 - 2015) la saga de referencia. Actualmente este género de juegos se entiende de otra forma, aunque las intenciones del texto siguen teniendo sentido para el objetivo que se propone.

De esta forma, propusieron hacer de la ciudad entera un sistema multiagente, dividido en cuatro elementos esenciales: la organización, los agentes, el entorno y las interacciones.

La organización divide la gestión de la ciudad a lo largo de cuatro niveles. De más alto a más bajo están: la ciudad, que percibe todo lo que pasa dentro de ella y se encarga de mantener el equilibrio sin restringir al jugador en lo que quiere hacer; la capa de control que se encarga de gestionar los distintos recursos de la ciudad, como el controlador de tráfico, el de peatones, policías...; los *NPCs* en sí mismos, que actúan de forma inteligente y conforme a unos arquetipos que el jugador puede entender e interpretar; y otras entidades menores, que sirven para enriquecer más la experiencia pero cuya presencia no es imprescindible, como los animales.

Las interacciones definen cómo se comunican los agentes con su entorno y entre sí. Con el entorno, se pueden adoptar modelos en los que los objetos con los que interactúan los agentes transfieren su información a los agentes, o ir más allá y hacer que la interacción con objetos nuevos esté condicionada por el conocimiento que poseen de haber interactuado con objetos similares en el pasado.

Lamentablemente, hacer de una ciudad un sistema multiagente tendría un coste computacional desmesurado, ya que todos los agentes de los cuatro niveles estarían evaluando su conocimiento continuamente. Sin embargo, propone una solución a este problema inspirándose en un concepto nacido en la informática gráfica, el *nivel de detalle*.

Con esta idea propone su *LOD Architecture*, compuesta por tres niveles: *Mastermind*, *overlords* e individuales. Los individuales son agentes que son totalmente funcionales, comportándose regularmente y siendo actualizados al máximo ratio. Cuando el sistema cree que un agente deja de ser importante, lo degrada y de su gestión se encarga un *overlord*. Ese *overlord* es en realidad un agente más, un controlador, que gestiona y decide qué información es relevante para los agentes degradados que están a su cargo, y qué acciones deben afectar al entorno. El *Mastermind* representa el menor nivel de *LOD*, dirigiendo a los *overlords* y decidiendo cómo actualizarlos.

De esta forma, los agentes cuyo comportamiento no sea directamente apreciable por el jugador son degradados, ahorrando así recursos que pueden ser destinados a otros intereses.

2.3.3. *Narrative BDI*

Revisiting Character-Based Affective Storytelling under a Narrative BDI Framework (Peinado, Cavazza, y Pizzi, 2008) expone el problema que existe a la hora de desarrollar historias conducidas por personajes capaces de reproducir comportamientos humanos, ya que la mayoría de investigaciones se enfocan en modelar el razonamiento humano teniendo la aplicación a la narrativa como algo secundario, alejándose de los problemas reales del fenómeno narrativo.

Apoyándose en otros estudios, defiende que es necesario lograr un compromiso entre las orientaciones narrativa y cognitiva para construir un *storytelling* complejo y profundo, para la que proponen la teoría cognitiva *BDI* (*Beliefs, Desires, Intentions*) como forma de unificar ambos pilares.

Su propuesta, *Narrative BDI*, modela el conocimiento no en función de profundos estados mentales o físicos, sino de estados emocionales, a los que denominan *emotional operators*. Éstos operadores emocionales se dividen en tres categorías: Operadores de Interpretación, que actualizan los sentimientos en respuesta de cambios en el entorno; Operadores de Interacción, que modifican directamente las creencias de otros personajes; y Operadores Físicos, que modifican las posiciones de los personajes en el mundo.

En su aproximación a la mejora del *storytelling*, se establece que sólo los personajes relevantes requieren una formalización BDI auténtica. Las creencias y sentimientos son modificados por los efectos desencadenados por la percepción de operadores emocionales, las acciones atómicas de la representación; los deseos y las metas son actualizadas en función de las emociones del personaje, que son los hechos atómicos de la representación; y las intenciones están divididas en aquellas que pueden ser parte de un plan racional o aquellas que son sólo reacciones.

Ambas pueden estar únicamente compuestas por operadores de interacción y operadores físicos, ya que los operadores de interpretación no son controlados por los personajes, dado el carácter de las emociones, que no responden a razonamientos lógicos. De esta forma, es el planificador situado sobre los personajes el encargado de disparar los operadores de interpretación de cada personaje en función de su personalidad predefinida.

En definitiva, *Narrative AI* proporciona una interpretación del *BDI* en la que no todo lo intrínseco a los personajes es controlado y utilizado en su razonamiento por ellos mismos, escapándose de su control sus emociones, que son controladas por un planificador, obedeciendo la naturaleza de cada personaje.

2.3.4. *GoldMiners*, caso de estudio de Jason

Goldminers fue el nombre dado al tema del VII Concurso CLIMA (*Computational Logic in Multi-Agent Systems*) organizado en 2006. En él se proponía a los participantes realizar un sistema multiagente capaz de coordinar a un equipo de cuatro mineros que se enfrentaría a otro equipo igual, con el objetivo de ver qué quienes eran capaces de recolectar más rápido los lingotes de oro repartidos por el escenario virtual. Los autores de la versión de Jason describen su implementación en el manual oficial de la herramienta (Rafael H. Bordini, 2007).

El entorno del concurso era implementado por los organizadores en un servidor remoto que simulaba la mina de oro, mandando percepciones sobre el entorno a los agentes y recibiendo peticiones de acción por parte de éstos, de acuerdo a un protocolo fijado en las bases del concurso. La implementación de Rafael H. Bordini y Jomi Huebner hizo uso de la personalización de arquitecturas y uso de *internal actions* que ofrece su propio intérprete, Jason, para llevar a cabo el razonamiento de forma remota.

El diseño de la estrategia de equipo consistió en la creación de dos roles: el de minero, que sería tomado por cuatro agentes cuyo cometido sería encontrar oro y llevarlo a un depósito; y el de líder, encargado de asignar cuadrantes —de los cuatro en que divide el mapa— a cada minero según su proximidad, y asignar piezas de oro encontradas a los mineros libres —que no están transportando oro—, eligiendo para cada pieza al minero más cercano.

Cada minero recorre de izquierda a derecha, verticalmente, su cuadrante asignado. Cuando un minero encuentra una pieza de oro y está libre, se propone coger esa pieza de oro. Anuncia que se dirige a ella a los demás mineros, y cuando llega a la posición del oro, lo coge y se lo comunica a los demás. Vuelve al deposito a dejar el oro y se propone como objetivo otra pieza de oro, en caso de que hubiera alguna no asignada a ningún otro minero. Y si no lo hay, recorre su cuadrante.

Si un minero encuentra una pieza de oro pero ya carga con una, comparte con los demás su ubicación. De este modo, todos los mineros libres envían al líder una puja con la distancia a la que se encuentran de la ubicación, y los mineros ocupados mandan una distancia cota superior, de forma que cualquier puja sea siempre menor que esa. Cuanto menor es la distancia, mejor es la puja. El líder comparte con todos los mineros a quién se le asigna el oro, haciendo que el elegido se mueva hacia él, y el resto sigan recorriendo sus cuadrantes.

Por último, si un minero se dirige hacia una posición con oro, pero por el camino encuentra una pieza de oro, abandona el deseo de coger el primero, anuncia que ahora no va a cogerlo —provocando que el resto de mineros libres pujen por él—, lo recoge y lo lleva de vuelta al depósito.

Esta implementación le concedió a sus diseñadores la victoria en el concurso.

Capítulo 3

Objetivos y especificación

El propósito principal de este trabajo, como se adelanta en la Introducción, es explorar el uso de modelos cognitivos avanzados en el desarrollo de personajes para videojuegos. Se busca poder aportar una capa de interés adicional a la jugabilidad, al dotar de comportamientos más inteligentes a los personajes, aportando inmersión y credibilidad, generando más proactividad —lo que propicia la aparición de *Jugabilidad Emergente*—.

La idea de este trabajo se inspira en la manera en que se plantea la IA de los personajes de un parque temático de la serie de ficción *Westworld* (Jonathan Nolan, 2016). Para cada agente no hay un guión explícito, sino que es construido por un "director" a base de ensayo y error. En ese proceso, se tiene en cuenta cómo el comportamiento del androide está condicionado por el modo en que sus aspiraciones y obligaciones (estas sí, explícitas) chocan con los cambios en su entorno, haciéndoles improvisar, creando nuevos desenlaces para cada "partida", etc. Expuesto esto, la extrapolación de este modelo a los videojuegos de mundos abiertos persistentes es inmediata. Para otros géneros también se presume útil, dado el ahorro en crear estructuras complejas para definir el comportamiento de los *NPCs*.

Para lograr dar forma a esta idea, se decidió utilizar la arquitectura multiagente basado en *BDI* que ofrece *Jason*. De este modo, se modelaría el conocimiento y los razonamientos de los personajes, los cuales podrían interactuar con el entorno, implementado en Unity.

3.1. Objetivos

Para dar forma a la idea de este trabajo, se propusieron tres grandes objetivos.

- Comprender y valorar lo que puede ofrecer la aplicación de *Jason* a videojuegos. Se tratará de responder a la pregunta sobre la utilidad de aplicar este tipo de sistemas multiagente frente a otros sistemas.
- Desarrollar un conector que una el sistema multiagente (*Jason*), con el entorno de desarrollo de videojuegos *Unity*, y así comunicar la *mente* y el *cuerpo* de cada personaje, implementados en respectivas herramientas. Posteriormente, y sobre este conector, diseñar y prototipar un videojuego que sirva para ilustrar el potencial del paradigma *BDI*.
- Probar dicho juego con usuarios reales para asegurar la usabilidad del mismo y la satisfacción del usuario, además de validar si la inclusión de comportamientos inteligentes es relevante desde el punto de vista de la experiencia del jugador.

Los resultados del primer y tercer objetivo se detallan en el Capítulo 6, mientras que el desarrollo del segundo se expone en el Capítulo 5.

3.2. Especificación de requisitos

Esta sección se divide en otras dos. La primera detalla los requisitos a cumplir por el conector entre Unity y Jason, denominado *UniJason*, y en la segunda se especifica el documento de diseño del juego *Miners of the Broken Planet*, el prototipo creado para ilustrar la utilidad del sistema.

Los requisitos que a continuación se enumeran son el resultado final de varias iteraciones, dada la naturaleza ágil de la metodología empleada en el proyecto.

3.2.1. Conector entre Unity y Jason, *UniJason*

Antes de nada, y a fin de entender los requisitos posteriormente enumerados, se debe explicar brevemente cómo interactúan los agentes con el modelo en la propia implementación de Jason. Los agentes constantemente piden al entorno, programado en Java, la ejecución de una acción cualquiera, y el entorno les informa de si se ha realizado o no con éxito dicha acción. Estas peticiones se realizan de forma concurrente entre los agentes.

Para ello deben cumplirse las siguientes funciones:

- El conector no puede contener ninguna información sobre el modelo implementado en Unity. Todo lo concerniente al tratamiento de los cambios del entorno de juego deberán ser tratados en la implementación ASL de los agentes, o en la implementación del modelo en Unity, según corresponda.
- Al tener que ejecutar los agentes acciones que pueden fallar o no, debe asegurarse que para cada acción solicitada al entorno se reciba una respuesta.
- Dada la naturaleza concurrente del funcionamiento de los agentes, debe asegurarse que las respuestas a las acciones solicitadas sean recibidas por aquellos que las solicitan y no por otros.
- De igual manera se deben poder enviar peticiones que necesitan respuesta para cálculos concernientes a *Internal Actions* que puedan ser implementadas en Java.
- Se deben poder recibir mensajes que no necesiten respuesta a ambos extremos del conector. Tanto mensajes de información que sean enviados a Unity, como nuevas creencias enviadas a Java/Jason, por ejemplo.

Además *UniJason* debe poder ser utilizado por cualquier videojuego, es decir, debe ser genérico y fácilmente extensible.

3.2.2. Prototipo de videojuego, *Miners of the Broken Planet*

Se trata de un juego de estrategia en tiempo real en el que el jugador debe coordinar a sus tropas para conseguir menas de Aleph en el menor tiempo posible. Además, el jugador tendrá que tener en cuenta la personalidad de cada personaje.

Aspectos Generales

Tiene lugar en el universo del juego de acción multijugador recientemente estrenado *Raiders of the Broken Planet* (Mercury Steam, 2017), en el cual, cuatro facciones se disputan el control del Planeta Roto por la explotación de una sustancia, llamada Aleph, que sólo se encuentra allí.

Este homenaje al juego original pone al jugador en la piel de un comandante encargado de coordinar a cinco agentes desplegados en ese planeta, para que encuentren suficientes menas de Aleph antes de que los escáneres enemigos les localicen. El objetivo último sería construir un juego completo, con varios niveles, aumentando la dificultad incrementalmente a base de reducir el tiempo disponible y/o las menas de Aleph a conseguir. Para este prototipo será suficiente con un único nivel.

El jugador asignará antes de jugar el nivel una clase de las tres disponibles a cada agente. Las clases son:

- *Soldado*: Tiene el movimiento más lento de todos, pero es la única clase capaz de matar enemigos.
- *Recolector*: Es la única clase capaz de extraer Aleph.
- *Explorador*: Tiene el movimiento más rápido de todos y es la única clase capaz de marcar posiciones exactas de filones de Aleph, una vez los encuentra. Las demás clases sólo son capaces de indicar cuadrantes peligrosos (por haber visto un enemigo) o prometedores (por haber visto una mena de Aleph), pero no posiciones exactas de filones de Aleph.

De forma paralela, el jugador deberá descubrir la personalidad de cada personaje, pues esta afecta de forma distinta según la clase asignada, haciendo que los niveles más complicados del juego sean imposibles de superar si no se asignan las clases de la forma menos perjudicial posible para el éxito de la misión, teniendo en cuenta las personalidades.

Es un juego de perspectiva cenital, pudiendo contemplar parte del mapeado y las posiciones de los agentes en él. El terreno de juego se divide en 9 cuadrantes formados por casillas. En el caso del prototipo, son cuadrantes de 12x12 casillas.

Menús y modos de juego

El juego contará con un menú principal desde el que se podrá acceder a una Nueva Partida o Salir del juego. Será necesario un tema musical, el tema principal, y el logo del videojuego. Sólo existe un modo de juego, que es el único descrito en este documento.

Configuración

Las opciones de configuración se limitan a las ofrecidas en el lanzador de la versión ejecutable del juego.

Interfaz y cámara

La vista del juego es cenital. Los personajes se distinguen por una luz de un color distinto según su clase: azul para soldado, rojo para explorador, y verde para recolector.

Como componentes del HUD (*Head-Up Display*) tendremos:

- El retrato del personaje seleccionado.
- Una cámara que muestra en tiempo real lo que hay cerca del personaje seleccionado.
- Una lista de nombres para seleccionar a un personaje por teclado.
- Un log con la información que los personajes le dicen al jugador.
- Un contador de cuenta regresiva en la parte superior central.
- Un texto que informe al jugador de que ha ganado o perdido la partida.

Las maneras de seleccionar al personaje a manejar son vía ratón, pulsando el icono del HUD correspondiente, o directamente con las teclas 1-5 del teclado. El personaje seleccionado muestra un parpadeo intermitente en su luz. En la versión final, entre nivel y nivel, aparecería un menú en el que se mostrarían los objetivos de la siguiente misión, y los retratos de cada personaje y las clases que podemos asignarles. Empezará el nivel una vez se confirmen esas asignaciones.

Jugabilidad

En esta sección se define la jugabilidad del prototipo. Su explicación se realiza a través del modelo MDA (*Mechanics-Dynamics-Aesthetics*), que separa su análisis en mecanismos, dinámicas y estéticas.

Mecanismos

Son las acciones más básicas: Las reglas del juego, cada acción simple que el jugador puede hacer, la representación de la información en pantalla, etc.

Mecanismos de interfaz El *log* de texto se llena con información recibida de los personajes durante la ejecución del juego. En él se vuelca la información que dan de lo que ven, de lo que hacen, etc.

Cada personaje tiene una luz de un color concreto sobre él, para facilitar su localización al usuario durante la partida. Cuando se selecciona a un personaje, éste mostrará una luz intermitente, entre el color original de su luz y el color blanco, para mostrar que es a él a quien se ha escogido. También cambiará, en el HUD, el retrato mostrado y la imagen de la cámara se actualizará para mostrar al personaje en tiempo real.

Mecanismo de control En primer lugar, están aquellos mecanismos relacionados con el control de los personajes. Al seleccionar uno, si se coloca el cursor sobre un cuadrante, éste se iluminará, y si se da *clic*, el personaje irá hacia él. Si aparece en la interfaz un indicador en una posición concreta y se selecciona a un personaje recolector, es posible mandarle a esa precisa ubicación para extraer el Aleph. De este mecanismo dependen los de exploración, ataque y recolección que se explican más adelante.

Mecanismo de visión y exploración Otro mecanismo sencillo es el visión. Si un personaje ve un enemigo en un cuadrante concreto, éste se marcará de rojo. Si por el contrario ve Aleph, lo marcará de color dorado. Si además, quien lo ve es un *Explorador*, no marcará todo el cuadrante, sino que generará un indicador señalando la posición concreta.

Movimiento de enemigos Los enemigos están confinados a un sólo cuadrante, pudiéndose mover únicamente por él, recorriendolo de izquierda a derecha y de arriba a abajo.

Mecanismos de ataque Los enemigos matan a cualquier personaje con el que se encuentren en la misma casilla, a no ser que sea un *Soldado*, y además al personaje que vean, lo persiguen hasta que dejan de verlo. Los soldados son la única clase capaz de matar enemigos, y lo hacen de la misma forma.

Mecanismo de recolección Los *Recolectores* extraen Aleph si lo encuentran ellos mismos. Esto puede ocurrir si se están dirigiendo a un Aleph más alejado y deciden coger el que acaban de ver, o si simplemente están explorando un cuadrante.

Las diferencias entre cada clase se resumen en el Cuadro 3.1.

Característica \Clase	Soldado	Recolector	Explorador
Percepción	2 casillas de distancia	1 casilla de distancia	2 casillas de distancia
Invulnerable	Sí	No	No
Velocidad	Lento	Normal	Veloz
Anuncio de Eventos	Por cuadrante	Por cuadrante	Por cuadrante para enemigos Por casilla para vetas

Cuadro 3.1: Diferencias entre Clases

Al Sistema de Clases se le añade una capa extra de complejidad, el Sistema de Personalidades. Al igual que ocurre con el de clases, cada personaje tendrá una. Éstas pueden ser *Neutral*, *Cobarde*, *Disidente* y *Traidor*. Este sistema no se puede entender sin el Sistema de Clases, por eso se detalla ahora y no antes. Las implicaciones que tiene cada personalidad para cada clase se describen en el Cuadro 3.2. Aclarar que estas implicaciones no son permanentes, sino que tienen una probabilidad de manifestarse.

Personalidad \Clase	Soldado	Recolector	Explorador
Neutral	No cambia	No cambia	No cambia
Cobarde	No mata aliens	Anda más despacio por cuadrantes peligrosos	No cambia
Disidente	No anuncia cuadrantes con vetas de Aleph	No cambia	No anuncia casillas con vetas de Aleph
Traidor	No cambia	Roba vetas de Aleph	Anuncia que hay enemigos donde hay Aleph

Cuadro 3.2: Cambios en Clases según personalidad

La existencia de estas personalidades en principio es desconocida para el jugador, aunque sus efectos sí puedan percibirse. La idea es que, en una versión completa del juego, se fueran obteniendo pequeñas pistas sobre qué personalidades existen y cómo pueden afectar al éxito de la partida.

Asignación de personalidades En cada partida, sólo dos personajes dispondrán de personalidad *Neutra*. El juego reparte las otras 3 personalidades entre los demás personajes, sin informar al jugador de quién posee cuál.

Dinámicas

Las dinámicas, dentro del *framework* MDA, definen el comportamiento de los mecanismos en relación a lo que hace el jugador, y cómo cooperan éstos mecanismos entre sí.

Para ganar, es necesario recoger un número concreto de menas de Aleph antes de que el contador llegue a 0. El jugador debe optimizar el tiempo que invierten sus mineros en encontrar y extraer Aleph, coordinándolos de forma eficiente.

Dinámicas asociadas a las clases Principalmente, el *Soldado* será utilizado para despejar cuadrantes de enemigos, y así hacerlos seguros para el resto de clases que no son invulnerables. Para los *Exploradores*, el jugador los destinará a cuadrantes inexplorados o en los que se sabe que hay aleph, para marcar posiciones exactas. Esto es importante, puesto que así puede enviar a sus Recolectores a que recojan Aleph y regresen inmediatamente después, reduciendo el tiempo de exposición a posibles enemigos. El jugador puede utilizar a los *Recolectores* para recoger Aleph directamente en posiciones ya marcadas por exploradores, o puede enviarlos a buscar aleph en cualquier cuadrante.

Dinámicas asociadas a las personalidades Al jugador le interesa descubrir qué personalidad tiene cada personaje, puesto que, como se ha mostrado en el apartado de *Mecanismos*, cada una afecta de forma distinta a cada clase. De esta manera, será importante saber a quién no asignar ciertas clases. La manera de inferir esas personalidades, es a través de la observación de comportamientos mostrados anteriormente en el Cuadro 3.2.

Estéticas

La cámara está ahí para dar más dinamismo a la partida, y que ésta no sea sólo ver puntos moverse por un plano.

El contador regresivo estará formado por minutos, segundos y milisegundos, y la música irá acelerando su ritmo según quede menos tiempo. De esta forma se pretende conseguir acrecentar la sensación de urgencia durante la partida.

Relato breve y parcial de una partida

Antes de cada partida, el jugador reparte las clases entre los personajes como vea conveniente, aunque al menos un personaje debe ser de clase recolector. En este caso, pongamos que elige 1 soldado, 2 recolectores y 2 exploradores. Los 5 personajes empiezan en el cuadrante del centro, así que el jugador manda a explorar a dos exploradores por los cuadrantes este y oeste. El explorador del este indica que ha avistado un enemigo (ese cuadrante se tiñe de color rojo), así que el jugador envía al soldado a dicho cuadrante, con la esperanza de acabar con el enemigo. Mientras, el explorador del oeste ha encontrado 3 filones de Aleph, así que el jugador envía a sus dos recolectores a por ello. Estos recogen una mena de Aleph cada uno (lo máximo que pueden cargar) y vuelven a la base con ellos, aumentando el contador en dos puntos. Como aún queda uno por recoger, ahora el jugador no arriesga personal y manda sólo a uno de ellos. El soldado, mientras tanto, ha encontrado y ha matado al enemigo que merodeaba por aquel cuadrante peligroso. El jugador seguirá avanzando así hasta conseguir la puntuación objetiva, ganando la partida.

Contenido

En este apartado se comentan los demás apartados que dan forma al juego: El trasfondo del videojuego y el diseño de los niveles.

Historia La acción se desarrolla en el Desierto de Sargon, un lugar famoso por sus constantes tormentas de arena. En él se encuentra un regimiento de la División Hades buscando yacimientos superficiales de Aleph. Estos yacimientos son muy preciados debido a que el Aleph puede ser extraído directamente de ellos, sin necesidad de construir grandes perforadoras.

Durante los primeros días hubo algunas bajas, pero los soldados lo achacaron a las tormentas de arena, en las que perderse era extremadamente sencillo. Sin embargo, con el paso de los días, algunos empezaron a hablar de un demonio cazahombres que habitaba en aquel desierto.

Este trasfondo está basado en la historia de la *skin* "Emnu Dagan" del personaje Shae. En él se cuenta que Shae tomaba la apariencia de un demonio del folklore nativo para matar y aterrorizar a los soldados humanos, con el fin de que se marcharan del Planeta por miedo a una amenaza sobrenatural. El jugador tomará el control de soldados de la División Hades (los malos), que tendrán que recolectar Aleph intentando evitar que Shae les cace.

Niveles En la versión final habría varios niveles que cambiarían el mapeado en extensión, obstaculos, y posiciones en las que hay Aleph. El prototipo desarrollado aquí sólo contará con un nivel, que se caracteriza por tener la misma configuración que el mundo 3 de *GoldMiners*, como se muestra en la figura

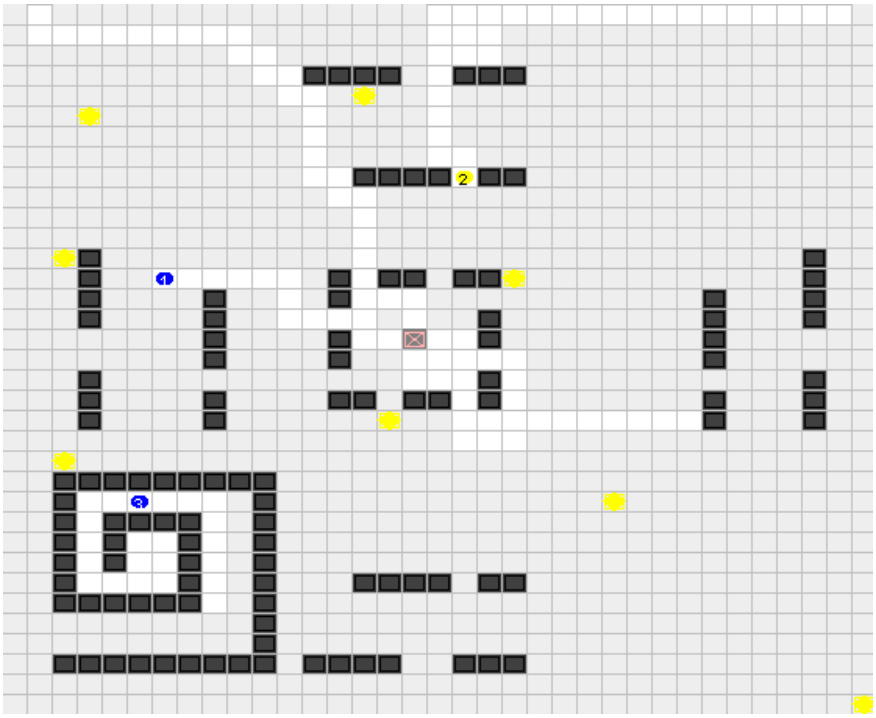

Figura 3.1: Distribución de obstáculos (de color gris oscuro)

Capítulo 4

Metodología y herramientas

En este proyecto se ha seguido una Estructura de Descomposición de Trabajo muy sencilla, cuyo trabajo se distribuiría a lo largo de tres hitos durante el curso. La metodología seguida ha sido ágil, realizando periódicamente reuniones para mostrar progresos, discutir cambios y marcar nuevas metas.

4.1. Estructura de Descomposición del Trabajo

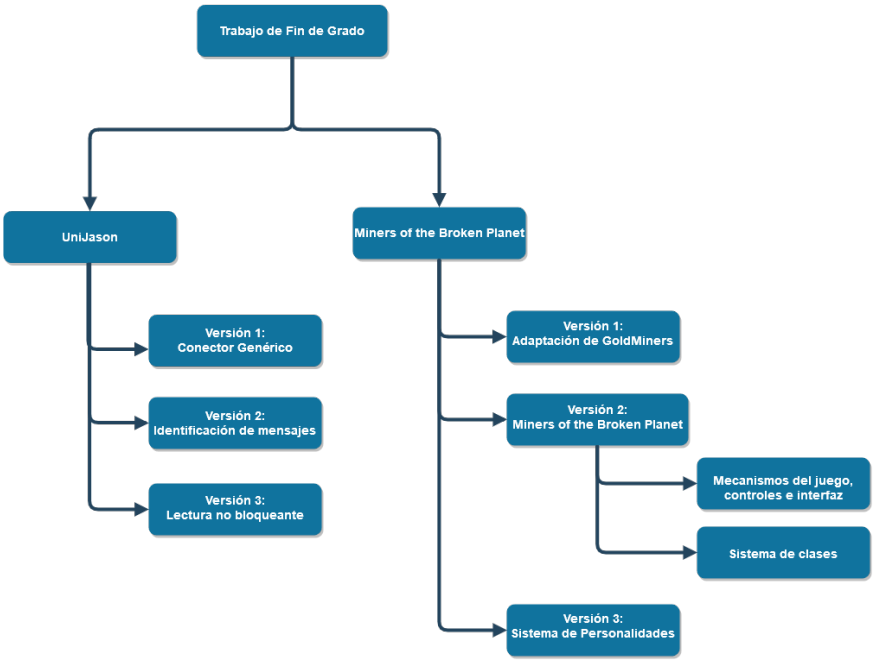

Figura 4.1: EDT seguida en el proyecto

Como se muestra en la Figura 4.1, ambas partes del proyecto fueron divididas en versiones. Cabe decir que las versiones de *UniJason* no estuvieron planificadas desde un principio, siendo éstas fruto de correcciones de errores relevantes que fueron descubriéndose a lo largo del proyecto. La versión 2 de *Miners of the Broken Planet* se descompone a su vez en el desarrollo del juego en sí y el de su sistema de clases.

4.2. Metodología

La metodología seguida ha sido una ágil propia, adaptando *Scrum* y Programación Extrema para un desarrollo de una sola persona. La razón de escoger una metodología ágil y no otra, es por la flexibilidad que necesitaba este trabajo para adaptarse a cambios de requisitos y problemas imprevistos. De hecho, dichos cambios y problemas se han dado, debido al aprendizaje progresivo de las herramientas a utilizar, a la par que se empleaban para desarrollar el TFG.

Por ejemplo, todo el aprendizaje del lenguaje AgentSpeak que utiliza *Jason* se ha hecho sobre la marcha, de manera que las posibilidades y restricciones del lenguaje se han ido descubriendo según se desarrollaba el proyecto. También el descubrimiento del funcionamiento de *Unity Engine* ha sido progresivo, así como el de las diferentes herramientas incluidas en él.

Para desarrollar el proyecto se ha avanzado de forma iterativa e incremental a través de *sprints*. El director principal del proyecto adoptó un rol de cliente a la hora de exigir nuevas características y validar las existentes, y también efectuó la labor de coordinador a la hora de fijar los objetivos de cada *sprint*.

4.3. Plan de trabajo

La idea inicial fue dividir el proyecto en dos partes, una por cuatrimestre. Así, el desarrollo de *UniJason* y la adaptación en *Unity* de *GoldMiners*, proyecto original de ejemplo de *Jason*, se realizarían durante el primero, y el prototipo propio durante el segundo.

Pero a principios del segundo cuatrimestre, durante una reunión con el director y el codirector, destinada a estudiar el tiempo y los recursos de los que se disponía para realizar un diseño de videojuego factible, se decidió que la segunda parte del proyecto, el prototipo, fuera una ampliación del *GoldMiners* original, aprovechando de esta manera la adaptación en Unity que se había comenzado a realizar en la primera parte.

Sobre la aplicación de la metodología, primeramente se realizaron *sprints* de dos semanas durante todo el curso, culminando cada uno en una reunión con el director del TFG, para comprobar el estado del desarrollo y proponer nuevas ideas de cara al siguiente *sprint*. Más adelante, y tras la finalización de la primera parte del proyecto, la concerniente a *UniJason*, cobró más importancia la labor del codirector, que al ser diseñador de juego en Mercury Steam, podía discutir el diseño del prototipo que constituye la segunda parte de este proyecto. A partir de este momento, los *sprints* adquirieron una duración algo más irregular, que oscilaba entre las dos y las tres semanas según la carga de trabajo.

Las reuniones con el director han sido en su mayoría presenciales, a excepción de algunas que se realizaron por Skype o Hangouts. Ambas aplicaciones permiten compartir pantalla, lo cual habilitaba al director a dar *feedback* directamente sobre el estado actual del proyecto.

Como se ha mencionado al principio de este capítulo, el desarrollo del trabajo final ha estado supeditado al cumplimiento de tres hitos a lo largo del curso. Estos hitos han coincidido con reuniones del grupo Narratech Laboratories, en las que los distintos equipos de TFG, TFM y Tesis exponíamos entre nosotros nuestros avances. Para la reunión del 21 de diciembre de 2017 se debía tener finalizada la herramienta *UniJason* y mostrar la adaptación del juego *Gold Miners* ya implementado en el entorno de desarrollo *Unity*. Para la reunión realizada el 22 de marzo de 2018 —previa a Semana Santa—, estando la segunda parte (*Miners of the Broken Planet*) ya en desarrollo, no hubo un objetivo claro, y sólo se mencionaron algunos avances concernientes a la implementación de la lógica de los agentes y algunos arreglos en el código entregado en el primer hito. Por último, se marcó el tercer hito para el 3 de mayo de 2018, en el cual los directores, codirectores y responsables de los distintos TFGs del grupo, probamos y proporcionamos *feedback* sobre éllos entre nosotros, produciendo así una lista de cambios a tener en cuenta de cara a las entregas finales.

La redacción de la versión final de la memoria empezó el 5 de mayo. Para su supervisión, se fijó el 16 de mayo de 2018 para entregar el primer borrador. Después, durante la corrección del borrador, se aplicaron los cambios propuestos en el *feedback* obtenido en la reunión del 3 de mayo de cara a la entrega final.

4.4. Herramientas de trabajo

Las herramientas que se utilizaron para el desarrollo del proyecto, la comunicación entre alumno, directores y colaboradores, el alojamiento de archivos, y la realización de la memoria fueron las siguientes:

4.4.1. Lenguajes y tecnologías utilizados

Principalmente, el trabajo se ha implementado en Unity Engine y Jason, pero estas herramientas a su vez se apoyan, en este caso, en otras a tener en cuenta.

4.4.1.1. Unity

Es uno de los motores de videojuegos más empleados actualmente. Es de propósito general, y para juegos tanto 2D como 3D. Es de uso gratuito para proyectos académicos. A continuación se comentan qué partes de Unity han sido utilizadas, en conjunto con Visual Studio, utilizado para la edición de los *scripts* de Unity.

La herramienta principal a destacar sería el *Canvas* de Unity. Permite agregar una capa a lo que captura la cámara principal de la escena, pudiendo añadir botones, paneles, ventanas de texto, imagenes, etc. En este caso, ha sido empleada en *Miners of the Broken Planet* para realizar su interfaz. A continuación, los elementos empleados, según *feature*, son:

- *Images*: 5 retratos, uno por cada personaje protagonista del prototipo.
- *Side Scroller* y *Text*: Un *log* deslizable en el que se escribe lo que dicen los personajes al jugador.
- *Text*: Un contador de cuenta regresiva.
- *Button*: Una lista de botones con los nombres de los personajes.
- *RawImage*, *RenderTexture* y *Camera*: 5 camaras que graban la cara de los personajes en tiempo real y lo proyectan en una textura, aplicada sobre una imagen.

También, se ha empleado *Animator*, para conocer las condiciones de activación y desactivación de las animaciones de los modelos 3D utilizados. Y por supuesto, el editor de escenas: Para la creación de *prefabs* (objetos personalizados compuestos de otros más pequeños o básicos), para el diseño de la escena de juego, y la colocación de todos los elementos que aparecen en la escena en una jerarquía limpia y ordenada.

4.4.1.2. Visual Studio

Es un entorno de desarrollo integrado que puede ser integrado en Unity para la implementación de scripts en C#. Se escogió esta herramienta por la familiaridad que se ha tenido con ella durante el transcurso de los estudios.

4.4.1.3. Jason

Es una librería de un interprete basado en Java de lenguaje AgentSpeak. Con el se pueden implementar sistemas multi-agente sobre una arquitectura BDI.

Sobre esta librería no hay mucha documentación. La mayor fuente de ésta la encontramos en su manual oficial, y como se cuenta en los apartados anteriores, a veces no cuenta todo lo que contiene la herramienta.

4.4.1.4. Jedit (Versión de Jason)

Jason cuenta con una versión de Jedit que permite prescindir de entornos de desarrollo para poder ejecutar código ASL. De esta manera, se emplea para ejecutar el codigo de forma sencilla, en conjunto con la *build* del juego creada desde Unity.

4.4.2. Comunicación

4.4.2.1. Skype y Hangouts

Este software permite realizar reuniones entre integrantes del grupo de forma remota. Se escogieron estas opciones ya que tanto director como autor disponían de ambas, y además permitían compartir las pantallas de nuestros ordenadores, facilitando el flujo de trabajo.

4.4.2.2. Slack

Esta herramienta de comunicación no es tan conocida pero es realmente útil. La usamos como recomendación del director del proyecto ya que él la utiliza para la comunicación con sus alumnos y con los distintos proyectos de los que se hace cargo. La ventaja que ofrece este sistema es que ofrece salas de chat organizadas por temas, dando la posibilidad de hacer tanto grupos privados como mensajes privados. Además tiene un buscador que permite acceder a todo el contenido de los grupos en los que estás inscrito o asociado. Otro punto fuerte de Slack es que integra una notable cantidad de servicios a terceros como pueden ser Google Drive, GitHub o Trello, entre otros.

Ha sido la herramienta principal con la que coordinarse con los directores para las reuniones cerradas, y con el resto de integrantes de Narratech para las reuniones de grupo.

4.4.3. Alojamiento compartido y control de versiones

Para el alojamiento del proyecto necesitábamos una buena plataforma de control de versiones, y un sistema general para compartir información.

4.4.3.1. GitHub

Es la herramienta que habíamos aprendido a usar de forma básica durante la carrera. Además permite hacer ramas del proyecto principal para poder desarrollar en paralelo sin riesgo a perder el contenido en caso de ser borrado localmente. En cualquier caso, las subidas a Github no han sido numerosas teniendo en cuenta que sólo ha trabajado una persona en este proyecto, pero sí han significado saltos enormes entre distintas versiones del trabajo.

4.4.3.2. Google Drive

Como sistema de alojamiento y compartición de ficheros utilizamos Google Drive, ya que su acceso es directo con la cuenta de la universidad. En él se han puesto en común anteriores trabajos finales, plantillas de documentos a incluir en la memoria, *assets* para el videojuego en Unity y otros recursos.

Se ha escogido esta herramienta sobre otros servicios de alojamiento en la nube debido a que se posee almacenamiento ilimitado gracias a las cuentas de la Universidad Complutense de Madrid, que tiene acuerdos con Google para ofrecer ventajas a sus profesores.

4.4.4. Realización de la memoria

Durante el desarrollo del trabajo, todas las notas fueron tomadas en papel. Finalmente, ha sido necesario elegir una buena herramienta para la redacción y maquetación final de la memoria, por lo que se propuso LaTeX, utilizando el editor en la nube Overleaf.

4.4.4.1. Overleaf

Se ha utilizado esta herramienta de escritura y edición de documentos como aplicación principal para desarrollar esta memoria. Overleaf es una herramienta *online* del conocido sistema de composición de textos LaTeX y que además permite la implantación de plantillas como por ejemplo la que está siendo usada.

Capítulo 5

UniJason y *MotBP*: Análisis, diseño e implementación

En este capítulo se aborda el análisis, diseño e implementación tanto de *UniJason* como de *Miners of the Broken Planet* (MotBP), en dos secciones distintas.

Para *UniJason*, en la sección 5.1.1 se analizará el TFG *IsoMonks*, cuál fue su planteamiento, qué hicieron y qué se puede reutilizar de cara a este trabajo. Además, se traslada todo lo concretado en la Especificación de Requisitos a conceptos más cercanos a la implementación. En el apartado de Diseño, se expone la arquitectura propuesta para el conector, y los diagramas de clases que la componen. En la Implementación, la propia implementación del conector, dividido en versiones.

En el caso del prototipo de videojuego, en el apartado 5.2.1 se concreta lo especificado en el Documento de Diseño de Juego. En el Diseño, se mencionan los distintos controladores que serán necesarios para la interacción con el juego, las clases que contienen el modelo del juego; y los diagramas de clases que relacionan entre sí las distintas clases del juego, y las clases de los controladores. Dentro de Implementación, las distintas versiones del juego: La adaptación a Unity de *GoldMiners*, *MotBP: Clases*, y *MotBP: Personalidades*.

5.1. Desarrollo de *UniJason*

Esta sección explica todo el proceso de desarrollo del conector *UniJason*.

5.1.1. Análisis

Se analiza tanto el TFG previo en el que se basa este trabajo, como la nueva especificación que detalla lo que debe implementarse ahora.

5.1.1.1. Análisis de *IsoMonks*

En *Isomonks* (Sánchez-López y cols., 2016), sus autores implementaron un conector entre Jason y *IsoUnity* (Pérez-Colado y Pérez-Colado, 2014), un plug-in para Unity para diseñar de forma sencilla juegos en perspectiva isométrica, para una demo concreta, inspirada en *La Abadía del Crimen* (Opera Soft, 1987). Como se contó en el Capítulo 2, a diferencia de este proyecto, ese conector no era genérico ni extensible, y estaba estrechamente vinculado a *IsoUnity*. Sin embargo, la idea en la que se cimentó dicho trabajo fue fácil de extrapolar al presente proyecto, y también su arquitectura ha servido para inspirar la de *UniJason*.

Por un lado, utilizaron mensajes en formato JSON para el envío de las acciones de los agentes (de Java a IsoUnity), y el envío de los cambios producidos en el modelo (de IsoUnity a Java). El parsing de dichos mensajes era explicito en ambos lados de la herramienta, es decir, no había una descomposición y tratamiento genérico de los mensajes, llegando a codificar explícitamente las dos únicas acciones que sus agentes podían hacer en su demostración en la parte de Java.

Aparte, al mandar acciones desde Java a IsoUnity, realizaban parte del tratamiento de cada acción concreta desde Java, no dejando que el modelo, que es quien debería poseer la información explicita de las consecuencias de esa acción en el entrono, responda al éxito o fracaso de esas acciones. Es decir, en su caso las acciones eran tan simples que se podía dar por hecho que siempre se cumplían, enviando la acción para que el modelo la representara, y sin importar si se realizaba con éxito o no en el modelo, desde Java ya se insertaban, en los agentes, las creencias de que esa acción había sido realizada con éxito.

Esto contraviene el valor genérico que se pretende alcanzar con este proyecto. Consideramos que la ejecución de las acciones por parte de los agentes, y la interpretación-realización de los cambios recibidos desde Unity, en Java/Jason, no pueden contener ninguna información acerca del modelo del videojuego/simulación que exista en Unity.

5.1.1.2. Análisis de la Especificación de Requisitos

A continuación, se analizan y comienzan a dar forma los requisitos especificados en el Capítulo 3.

Como se menciona en la especificación, es necesario que el conector no dependa del modelo del videojuego creado en Unity. Por esta razón, en el extremo de Java/Jason, hay que crear una clase genérica que extienda la original *Environment* de Jason. En ella se reimplementará el método *ExecuteAction* para que, por cada acción solicitada al entorno, se envíe esa petición a Unity, y espere a que Unity le responda si ha tenido éxito, o no. Dicha clase contiene los métodos necesarios para añadir, quitar y reiniciar las percepciones de los agentes, tanto a nivel global como a nivel particular.

En este punto, y antes de complicar más el análisis, es necesario esbozar el contenido de los mensajes que se enviarán desde Java y desde Unity. Jason accede al modelo en dos situaciones: cuando un agente solicita realizar una acción, y cuando un agente ejecuta una *InternalAction* que accede a información del modelo. Para ambos caso, hace falta una respuesta, y resultan separables en ámbitos diferentes, de modo que es lógico pensar que sea necesario distinguir entre dos tipos de mensajes a enviar o recibir: *de Entorno* y *de Información*. Esta diferenciación se materializa en forma de cabeceras en cada mensaje, para poder ser gestionado por separado, y por varios hilos en caso de ser necesario.

Como también se comenta en la especificación, no todos los mensajes son peticiones. Desde Unity se debe poder enviar mensajes que representen ediciones en la base de creencias de los agentes de forma directa, sin necesitar respuesta por parte de dicha base; e incluso desde Java debe poderse enviar información al modelo si se cree conveniente.

De este modo, desde el lado de Java/Jason, son etiquetadas *de Entorno*, por defecto, las peticiones de acciones de los agentes, y *de Información* las peticiones de información, en *InternalAction*s, sobre el modelo que se necesiten, dejándose a la implementación particular de cada videojuego. Desde el lado de Unity, son etiquetadas de Entorno, por defecto, las respuestas a las peticiones de los agentes y los distintos mensajes que emularán los métodos de *Environment*, y *de Información* las respuestas a las peticiones de las *InternalAction*s sobre información del modelo.

Al principio de este apartado se mencionaba que el entorno tiene dos ámbitos de modificación de la base de creencias: A nivel *Global*, y a nivel *Agente*. Por lo tanto, es imprescindible que los mensajes generados en Unity, para ser enviados al entorno genérico, puedan contener un campo que identifique al agente sobre el que realizar la modificación, cuando sean modificaciones a nivel *Agente*. Si el mensaje no posee el identificador de ningún agente, se entenderá que es una modificación a nivel *Global*. Este identificador también se utilizará en Java para denotar quién solicita una acción.

Entre los posibles mensajes que puede recibir e interpretar el entorno genérico, a efectos de *Entorno*, son cadenas que contienen los métodos implementados en *Environment* (*AddPercept*, *RemovePercept*, *RemovePerceptsByUnif*, *ClearPercepts*, *InformAgsEnvironmentChanged*); una etiqueta que indica que el modelo está listo para el arranque del sistema multiagente de Jason; y una etiqueta que indica que el contenido del mensaje es la respuesta a una petición de acción. Éstos mensajes son los únicos genéricos, dejando al programador la tarea de implementar los demás mensajes según convenga.

Con el propósito de unificar el envío y recibo de mensajes, y el tratamiento de éstos con cabeceras distintas, es conveniente implementar una única clase dedicada a ello, un conector externo, en cada extremo. En Java/Jason, ya que el envío de mensajes será concurrente, debe constar de un hilo que reciba constantemente mensajes, y los almacene en registros separados en función de su cabecera. Necesita métodos que accedan de manera sincronizada a dichos registros. Además, en el caso de las peticiones se necesita que cada una este firmada con un ID, haciendo que queden a la espera de recibir una respuesta que contenga ese mismo ID. En Unity, como el tratamiento de mensajes y la actualización de los cambios sobre el modelo es secuencial, el conector externo no necesita de mecanismos que sincronicen el acceso a ningún registro. Sin embargo, ésto no se llevaría a implementación hasta posteriores etapas del desarrollo, al no creerlo necesario en un primer momento.

5.1.2. Diseño

5.1.2.1. Diagramas de Clases

Diagrama de clases en Java

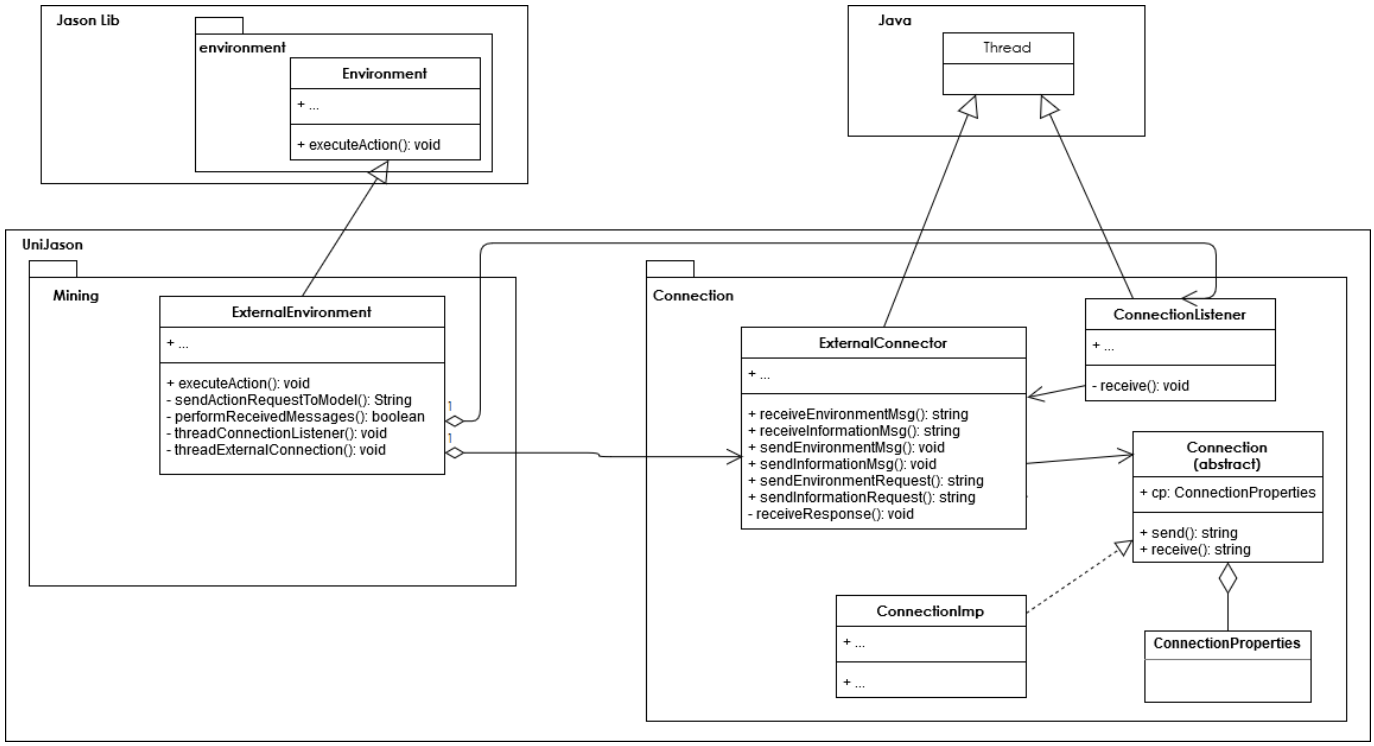

Figura 5.1: Diagrama de clases en Java

Como se aprecia en la Figura 5.1,se han distinguido tres dominios: El de Java, el de Jason y el de *UniJason*. De esta forma se detalla cómo *ExternalEnvironment* extiende la clase *Environment* de la librería de Jason, y cómo *ExternalConnector* y *ConnectionListener* lo hacen con la clase *Thread* de Java.

Diagrama de clases en Unity

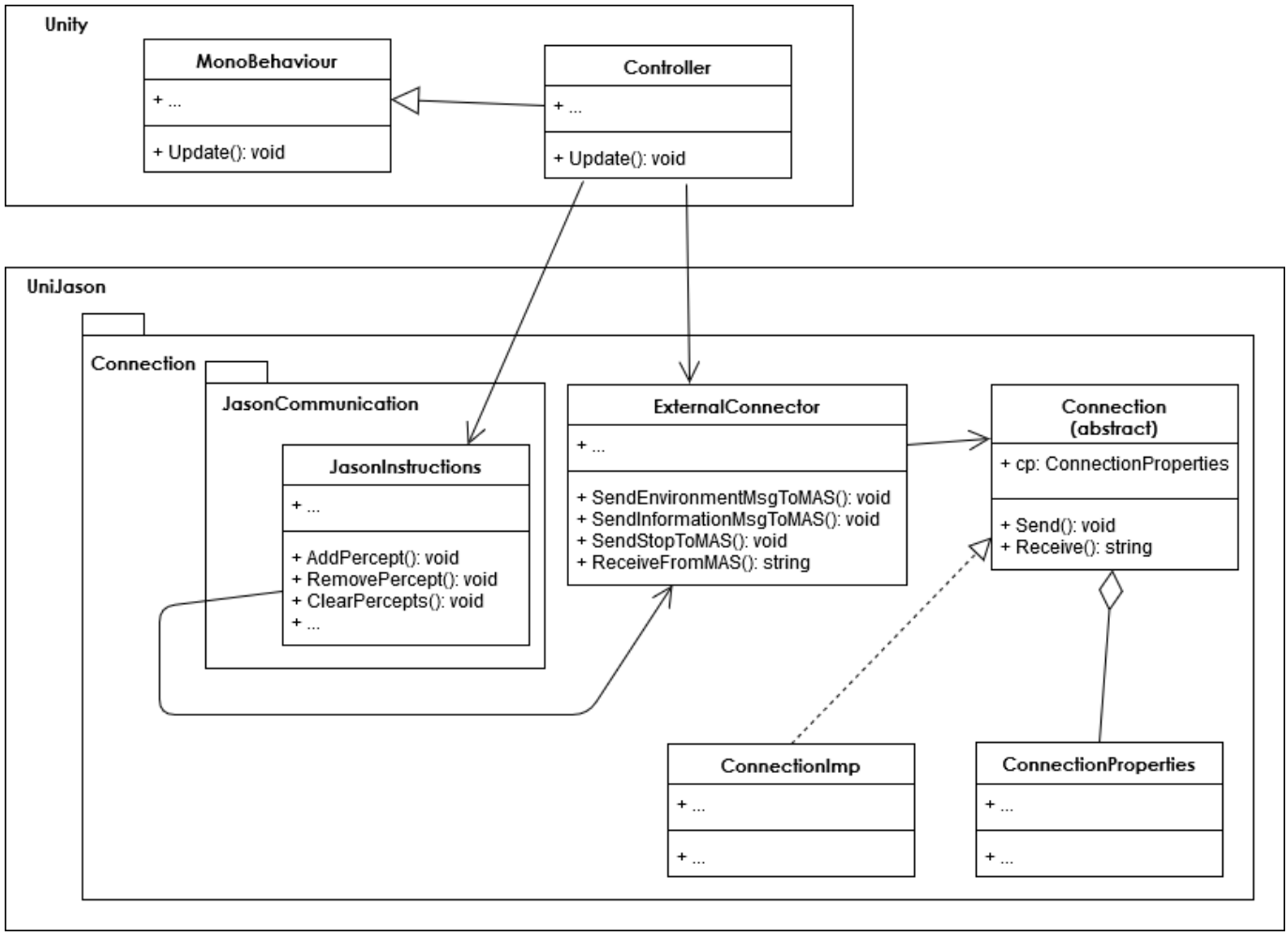

Figura 5.2: Diagrama de clases en Unity

Como muestra la imagen, el comportamiento de las clases de *Connection* es el mismo, con el añadido de *JasonInstructions*, que dispone de métodos para enviar información a Jason con el formato apropiado de los métodos de *Environment*. La clase *Controller* no es ninguna en particular, sino que representa que *ExternalEnvironment* y *JasonInstructions* están hechas para ser accedidas desde el método *Update* de cualquier clase que extienda de *MonoBehaviour* de Unity, que habitualmente será el controlador del juego.

5.1.2.2. Arquitectura del conector

En la Figura 5.3 se ilustra la comunicación entre los agentes y el modelo. El controlador, una vez más, simboliza el controlador que implementa en su método *Update* las llamadas a los métodos del conector externo.

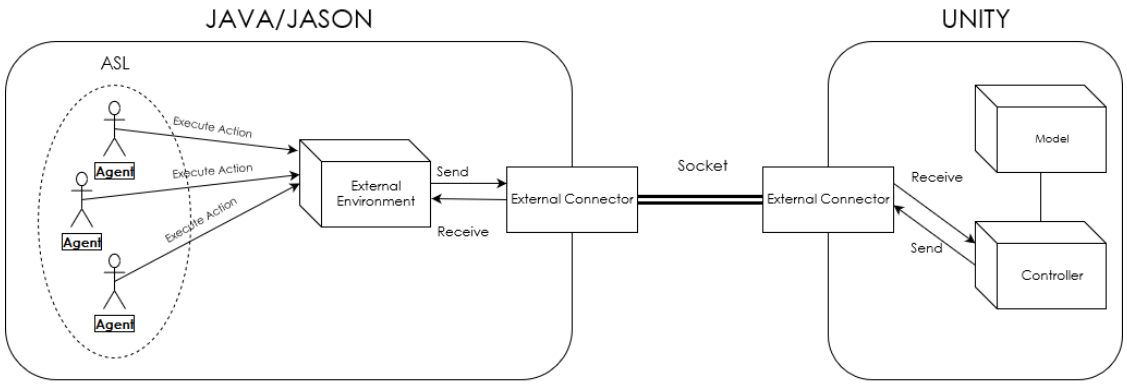

Figura 5.3: Arquitectura de la conexión entre Unity y Jason usando *UniJason*

5.1.3. Implementación

En este apartado se detallan las distintas versiones de UniJason: Una primera aproximación, con problemas que se arrastraron durante bastante tiempo, una revisión de la primera, logrando asegurar la fiabilidad del paso de mensajes entre ambos extremos, y una tercera versión que mejora el rendimiento en el extremo de Unity.

5.1.3.1. Versión 1: Controlador de Mensajes Genérico

Lo primero que se implementó fue el *parsing* de mensajes en ambos sentidos. En el extremo Java, con *ExternalEnvironment* se extendió la clase de Jason *Environment*, re-implementando el método *ExecuteAction*. Éste consistía en la construcción y envío de la petición de acción del agente en formato JSON, añadiendo el nombre del agente en un campo, *who*, que identifica quién solicita la acción. Justo después del envío, se queda esperando en un semáforo al *release* que se produciría desde otro método propio, *PerformReceivedMessage*.

También se re-implementó el metodo *Init*, que ejecuta dos hilos, de las clases estáticas *ConnectionListerner* y *ExternalConnector*, y queda a la espera de que Unity informe de que el modelo está listo. *PerformReceivedMessage* se encargaba de *parsear* los mensajes que llegan desde el modelo, primero a través del *ExternalConnector* y que luego pasan por el *ConnectionListener*, que ejecuta dicho método. En él se esperan recibir mensajes JSON cuyo nombre debía estar especificado en un enumerado de tipos de mensaje implementado en la misma clase.

En ese enumerado se describen los nombres de los métodos utilizables para modificar la base de creencias implementados en *Environment*; junto a *EnvironmentDone*, que es el nombre del mensaje que indica que el entorno está preparado para la ejecución de los agentes; y *ActionResponse*, que contiene un campo *success* indicando si la acción a la que responde se ha ejecutado con éxito.

En el procesado de este último tipo de mensaje, según el contenido de *success*, se cambiaba el valor de un registro compartido de la clase, y se hacía el *release* que se comentaba antes, permitiendo que el método *ExecuteAction* siga su ejecución, leyendo el contenido de dicho registro, y devolviendo *true* o *false* en función del valor de éste.

La clase *ExternalConnector*, que extiende de *Thread*, lee constantemente el *socket* en un bucle, y separa los mensajes recibidos según las etiquetas de ámbito que se comentaron en el Análisis, ahora representadas como "*(Environment)*" y "*(Information)*". La clase posee dos registros en los que guardar los mensajes recibidos según su ámbito. Al procesar el mensaje, lo guarda en un registro u otro, hace *release* del semáforo incluido en el método *ReceiveEnvironmentMsg*, o en el método *ReceiveInformationMsg*, según corresponda. A continuación, otro semáforo bloqueaba la recepción de mensajes, para no re-implementar el registro antes de que puediera ser leido. De esta forma, el *ConnectionListener* ejecutaba *ReceiveEnvironmentMsg*, quedando a la espera del *release* que confirmase que ya había un mensaje que leer para *parsear*, desbloqueando, cuando éso sucedía, la recepción de mensajes. Si el mensaje contiene la etiqueta de "*(STOP)*", el bucle acaba su ejecución y, con ella, la del hilo.

La finalización de la ejecución del hilo del *ExternalConnector* desencadena una serie de interrupciones de hilos. Justo después de que acabe el de ExternalConnector, se llama al *Stop* del ExternalEnvironment, que manda señales de interrupción tanto al *ExternalConnector* como al *ConnectionListener*, para que internamente interrumpan su propia ejecución. Finalmente, libera el puerto llamando a *StopConnection* de la clase *Connection*, que cierra el *socket*.

ExternalConnector utiliza los métodos *Send* y *Receive* de *Connection*, una clase abstracta implementada por *ConnectionImp*. ConnectionImp posee un atributo *cp* de clase *ConnectionProperties*, cuya constructora describe los atributos de la conexión: El puerto de salida (10000) y el de entrada (10001) de Unity, la dirección (*localhost*) y el tipo de socket (UDP). Al recibir los datos con *Receive* a través del *socket* se utiliza el puerto de salida de Unity, y al enviarlos con *Send* se usa el puerto de entrada de Unity.

En el extremo de Unity se creó un *ExternalConnector* similar, aunque sin semáforos, ya que el acceso a dicha clase se hace de forma secuencial y por un único hilo, en el que todos sus métodos están sincronizados. Se implementó una clase llamada *JasonInstructions*, que se compone de funciones que envían órdenes a través del el *ExternalConnector*, una por cada método del *Environment* de Jason (*AddPercept*, *RemovePercept*...) . De esta manera, sólo se tiene que incluir la creencia en forma de *string* al llamar a dichas funciones, y añadir otro *string* con el nombre del agente al que va dirigido, si es el caso.

De la misma manera que en Java, *ExternalConnector* usa los métodos *Send* y *Recieve* de *Connection*. La implementación en Unity es análoga a la de Java, con la excepción de que el atributo *cp* se inicializa la primera vez que se accede a la instancia de *Connection*. Ésta es creada, y acto seguido se inicializan las propiedades de la conexión (el atributo *cp*) a través del método *Initialize* de *Connection*. Cabe destacar que la implementación en esta versión del método *Receive* de *Connection* es bloqueante, lo cual acarreó problemas que se cuentan más adelante.

También hay que mencionar que *RemovePerceptsByUnif* no funciona siempre. Si desde Unity se envían creencias a eliminar por unificación que contienen "_", al convertir los mensajes de String a creencia en Java, con los propios métodos de Jason, se insertan números detras de cada "_" que imposibilitan su correcto funcionamiento.

5.1.3.2. Versión 2: Peticiones e Identificación de mensajes

Durante la implementación de *Miners of the Broken Planet* se cayó en un error fatal cometido al diseñar e implementar el conector. Desde Java, para cada petición de acción se recibía una única respuesta afirmando o negando su realización, pero no se aseguraba que el primero en solicitar la acción fuera el primero en llegar al semáforo. De esta manera, dada la naturaleza concurrente del sistema, un agente se podría colar y leer la confirmación de acción de otro agente cuya petición se hubiera escrito en el *socket* antes que la suya. Por supuesto, esto originaba fallos en el comportamiento de los agentes. De esta forma se decidió aprovechar este error de implementación para arreglarlo y de paso sofisticar la estructura del conector.

Según ésta versión, ahora el *ExternalConnector* posee dos métodos más, *SendEnvironmentRequest* y *SendInformationRequest*. En ámbos métodos, se genera un ID único para identificar al hilo solicitante, que interrumpe la ejecución hasta que el conector recibe un mensaje con identificador. Cuando ocurre, el conector lo notifica a todos los hilos que esperan uno, despertando comprobando cada uno si el ID del mensaje recibido es el suyo, y si lo es, devuelve el contenido del mensaje. De esta manera ahora, aparte de las etiquetas de ámbito, se acopla una etiqueta con el ID del solicitante. Se conservan los métodos *ReceiveEnvironmentMsg* y *ReceiveInformationMsg* para mensajes que no necesitan respuesta (y por lo tanto, tampoco identificador), y el *ConnectionListener* sigue funcionando igual.

En el *ExternalEnvironment* se prescinde de semáforos y registros compartidos entre hilos, y ahora, en lugar de ejecutarlo el ConnectionListener, es *ExecuteAction* quien ejecuta *PerformReceivedMessage* con la respuesta devuelta por *SendEnvironmentRequest*. De esta manera, el contenido del campo *success* de la respuesta no se escribe en un registro compartido entre hilos, y queda atrapado en el flujo de ejecución de *ExecuteAction*.

En Unity se hicieron los ajustes necesarios para responder a mensajes que contienen ID.

5.1.3.3. Versión 3: Lectura no bloqueante de mensajes

Durante las pruebas del 3 de mayo, se constató un problema de rendimiento del prototipo de *AlephMiners* por el cual la tasa de *frames* por segundo era mucho menor que en el equipo utilizado para el desarrollo. Gracias a la asistencia de uno de los integrantes de Narratech se dio con la raíz del problema: *AlephMinersWorld* llama cada frame al *CheckEvents* de *AMWorldUpdater*, que accede a través del *ExternalConnector*, y éste a través de *Connection*, al método *Receive* de *UDPClient* que identifica al servidor.

Receive bloquea el hilo de ejecución hasta que se recibe una respuesta por el *socket*, y como consecuencia alarga la duración de cada *frame*. Es decir, si no recibe mensajes lo suficientemente rápido, provoca una disminución en el desempeño del juego en Unity.

La causa de ésto es la implementación multi-hilo del sistema multi-agente de Jason, teniendo un hilo de procesamiento independiente para cada agente. De esta manera, como el procesado de los siete agentes del prototipo es más lento que en el equipo de desarrollo, cuya CPU dispone de ocho hilos, los mensajes desde Jason tardan más en llegar, alargando la espera de éstos en cada *frame*.

La solución propuesta por el compañero fue la de implementar una lectura asíncrona de mensajes, de manera que *Connection* leyera constantemente el *socket*, almacenando cada nuevo mensaje en un *buffer* al que accedería su método *Receive*, que devolvería el primer mensaje almacenado. *ExternalConnector* comprobaría cada *frame* si hay un nuevo mensaje, y si lo hay lo extraería con el método *Receive* de *Connection*. Si no lo hubiera, seguiría su ejecución normal, sin bloquear el hilo.

Se implementó esta opción haciendo uso de una cola, por la cual *Receive* devolvía los datos almacenados en ésta ordenadamente. Después se consideró e desarrolló otra alternativa, que consistía en utilizar la implementación original —la anterior a la idea del compañero— haciendo uso de una propiedad de la clase *UDPClient* que ofrece C# : *Available*. Esta propiedad proporciona el número de mensajes listos para ser leídos en el *socket*. Sólo bastó con añadir al método *Receive* de *Connection* una comprobación que comprobara la propiedad *Available* del *serverSocket*, para leer mensajes del *socket* únicamente cuando hay mensajes completos para ser leídos.

Se optó por esta alternativa puesto que el hilo principal de los *Update* de Unity realizaría esta misma comprobación para los mensajes de la cola, con el añadido de tener un hilo aparte leyendo constantemente el *socket* y cargando con los mensajes la cola, con el coste que eso supone.

5.2. Desarrollo de *Miners of the Broken Planet*

La elaboración del documento de diseño de juego en el que se basa el prototipo del videojuego fue posterior al desarrollo de la adaptación de *GoldMiners* para Unity. De este modo, aunque este apartado describa el desarrollo del prototipo, la importancia de *GoldMiners* y su adaptación hacen necesario que se hable de ellos. También cabe destacar que dicha adaptación no es interactiva como sí lo es la original, no permitiendo al usuario colocar piezas de oro en casillas concretas —lo cual era la única interacción en la original— para su posterior recolección por parte de los agentes.

5.2.1. Análisis

5.2.1.1. Análisis de GoldMiners

La implementación del *GoldMiners* original se lleva a cabo a lo largo de tres clases: *MiningPlanet*, que extiende a *Environment*; *WorldModel*, que contiene la información de la simulación, y *WorldView*, que implementa la vista. Claramente, presenta un patrón Modelo-Vista-Controlador, ya que *MiningPlanet* se encarga de gestionar la interacción que tiene el entorno con los agentes ASL, y viceversa.

La clase *WorldView* no es de relevancia, ya que la vista en la adaptación la pondrá el propio motor Unity. La clase *MiningPlanet* re-implementa el método *ExecuteAction*, el cual toma la acción recibida desde el ASL y la ejecuta a través de su método correspondiente, implementado en el modelo. Después de ésto, se actualizan las percepciones del agente implicado en la acción, a través del método *UpdateAgPercepts*, que borra las percepciones anteriores del agente y añade nuevas: Su posición actual y lo que se encuentra a una casilla de distancia de él. También se describe el método para la inicialización del sistema, que añade creencias globales, inicializa la vista, y actualiza las percepciones de todos los agentes, usando *UpdateAgsPercepts*, que llama a *UpdateAgPercepts* para cada agente.

WorldModel a grandes rasgos, contiene métodos que sirven para extraer información del modelo, y modificarlo. Extiende a *GridWorldModel*, clase que contiene la información del plano de juego en forma de matriz de datos bidimensional, además de sus métodos accesores y modificadores, y la codificación entera, en base 2, de los posibles objetos que puede haber en la matriz (*Clean* ->0, *Agent* ->2...). Entre los métodos relevantes del modelo, se encuentran aquellos que describen las acciones que los agentes pueden realizar: *Move*, y *Pick* y *Drop* para manipular el oro encontrado. Además, la información de cada mapa de la simulación está *hardcodeada* en métodos que llaman a otros de *GridWorldModel* para modificar la matriz de datos que contiene.

En vista de las éstas observaciones, se necesita la adaptación de las tres clases existentes importantes. Así, *GridWorldModel* es *GridWorld* en Unity, *WorldModel* es *GoldMinersWorld*, y *MiningPlanet* es *GMWorldUpdater*. En la adaptación, ya que *GMWorldUpdater* gestiona los cambios del modelo, es quien conecta directamente con la clase *ExternalConnector* de *UniJason*, por donde recibe y envía información relacionada con el modelo, contenido en *GoldMinersWorld*. Por tanto, en *GMWorldUpdater* se *parsean* las solicitudes, tanto de acción como de información, provenientes de Java, y se envían las correspondientes respuestas.

También se debe considerar que los agentes ejecutan el algoritmo A* —a través de la *InternalAction* implementada en Java *Get-direction*, de la biblioteca *jia*— para saber cuál es la próxima casilla hacia su destino, tomando información del modelo. La complicación surge cuando el modelo ya no está en Java sino en Unity, haciendo necesario el envío de la información requerida a través de *UniJason*. Ésta necesidad fue la que hizo necesaria la revisión del diseño de mensajes, creando la división en ámbitos que ya se mencionó en el apartado dedicado a *UniJason*, considerando el ámbito *Information* para este tipo de tareas.

5.2.1.2. Análisis del Documento de Diseño de Juego

Teniendo presente el anterior análisis, aquí se describen los cambios a realizar sobre la adaptación de *GoldMiners* para transformarlo en *Miners of the Broken Planet*, basándose en el *GDD*. Primero, se debería extender *GoldMinersWorld* y *GMWorldUpdater*, resultando *AlephMinersWorld* y *AMWorldUpdater*, respectivamente. Para *AlephMinersWorld*, habría que almacenar las clases del juego (Soldado, Recolector, Explorador) y las personalidades (Neutral, Cobarde, Disidente, Traidor), y asociarlas en alguna estructura para dar forma a cada personaje. También habría que añadir una nueva acción, *Kill*, para que tanto enemigos como soldados puedan eliminar del juego al agente objetivo. En el caso de *AMWorldUpdater*, habría que ampliar el rango de acciones a *parsear* para incluir la nueva acción (*Kill*), e implementar una manera de reflejar la información compartida por los agentes con el líder dentro de Jason, en el entorno de juego.

Adicionalmente, todo el videojuego estaría gestionado por controladores. Estos controladores deben gestionar las entradas por teclado y ratón, y tomar dichas entradas para modificar la interfaz de usuario y el entorno de juego. Añadir que las modificaciones en la interfaz giran en torno al personaje seleccionado en cada momento, y las del entorno de juego lo hacen alrededor de la visualización del mapa, variando sus colores e iconos. Dicho ésto, el control se dividiría principalmente en el controlador de ratón, el de teclado, el de selección de personajes, y el de recoloreado e iconos del mapa.

El controlador de teclado simplemente contendría un registro en el que almacenar la última tecla pulsada por el jugador, comprobando que ésta es alguna entre la teclas 1 y 5 (ó 0 para características que se explicarán más adelante), de manera que pueda ser comprobada por los demás controladores.

El controlador de selección de personajes comprobaría la tecla del personaje pulsada, modificando el retrato y la cámara mostrados, y estableciendo una luz intermitente sobre el personaje seleccionado.

El controlador de recoloreado e iconos dispondría de los métodos para colorear cuadrantes enteros del mapa de juego, en función de si el ratón está sobre él, o de si se ha recibido información de los agentes de que hay Aleph o enemigos en dicho cuadrante, o ya no hay nada de interés. También dispondría de métodos para colorear casillas concretas, y proyectar encima de ellas iconos sobre los que el jugador puede clicar. Sin embargo, no sólo gestiona el mapa real, sino también una capa que oculta el terreno de juego —apodada *Mist layer* por su similitud con una niebla de juego— situada entre la cámara de juego y el mapa real, que refleja los mismos cambios que el mapa real. La tecla 0 se utilizará para activar y desactivar dicha capa, aunque es una característica orientada al prototipo, que no estaría en el juego final.

Por último, el controlador de ratón se encargaría de comprobar sobre qué objeto interactuable está situado (cuadrante del mapa, icono de Aleph, o ninguno) para cambiar el diseño del cursor; y además comprobar, si estando sobre alguno, se ha clicado, para que se ejecute una función determinada. Poner el ratón sobre un cuadrante, hace que éste cambie su color a uno más claro, y clicar hace que el personaje seleccionado se mueva y explore dicho cuadrante. Poner el ratón o clicar sobre un icono de Aleph sólo tiene efecto si el personaje seleccionado es Recolector, cambiando el diseño del cursor y, si se clica, mandando al Recolector a esa posición para que recoja Aleph.

Hasta este punto, al mismo tiempo que se han propuesto los controladores del juego, ya se ha dado forma a los requisitos para gestionar la cámara y los retratos de la interfaz. Atendiendo al *GDD*, falta por hablar del resto de elementos de la interfaz: La gestión de los nombres de los personajes, el contador y el texto de fin de partida, y el *log* con la información compartida de los agentes. Todas ellas se detallan ahora.

Como de antemano se sabe que el número de personajes es fijo, se puede dedicar un espacio de la interfaz para colocar cinco objetos interactuables que contengan el nombre e información iconográfica de cada personaje, cuya interacción se manifiesta cambiando el personaje seleccionado. El contador no tiene complicación, resultando ser uno que se decrementa hasta llegar a 0, y dada su estrecha relación con el texto de fin de partida, es interesante concentrar todo en un mismo controlador. De esta manera, dicho controlador contiene el número de menas de Aleph necesario para ganar, y dispone de funciones que comprueban constantemente si ese número se ha alcanzado. Si no se alcanza y la cuenta llega a 0, muestra el mensaje de derrota. Si el jugador gana, la cuenta se para y se muestra una felicitación al jugador. Por último, para el *log*, se crea una ventana con *scroll* vertical en la que *AMWorldUpdater* escribirá los mensajes que el líder escuche, en Jason. Por lo tanto, en el lado de Java/Jason se necesita diseñar una forma de escuchar dichos mensajes y mandarlos a Unity.

Por supuesto, la implementación no queda ahí. También se modifica el código ASL de los agentes. Principalmente hay que restringir la autonomía que poseen los agentes en *GoldMiners* para realizar sus acciones, y crear nuevos planes para posibilitar la interacción directa con los agentes a través de Unity. También hay que crear nuevos comportamientos y modificar los existentes, para que se ajusten a la clase y personalidad de cada agente.

5.2.2. Diseño

5.2.2.1. Diagramas de Clases

Diagrama de clases en Java

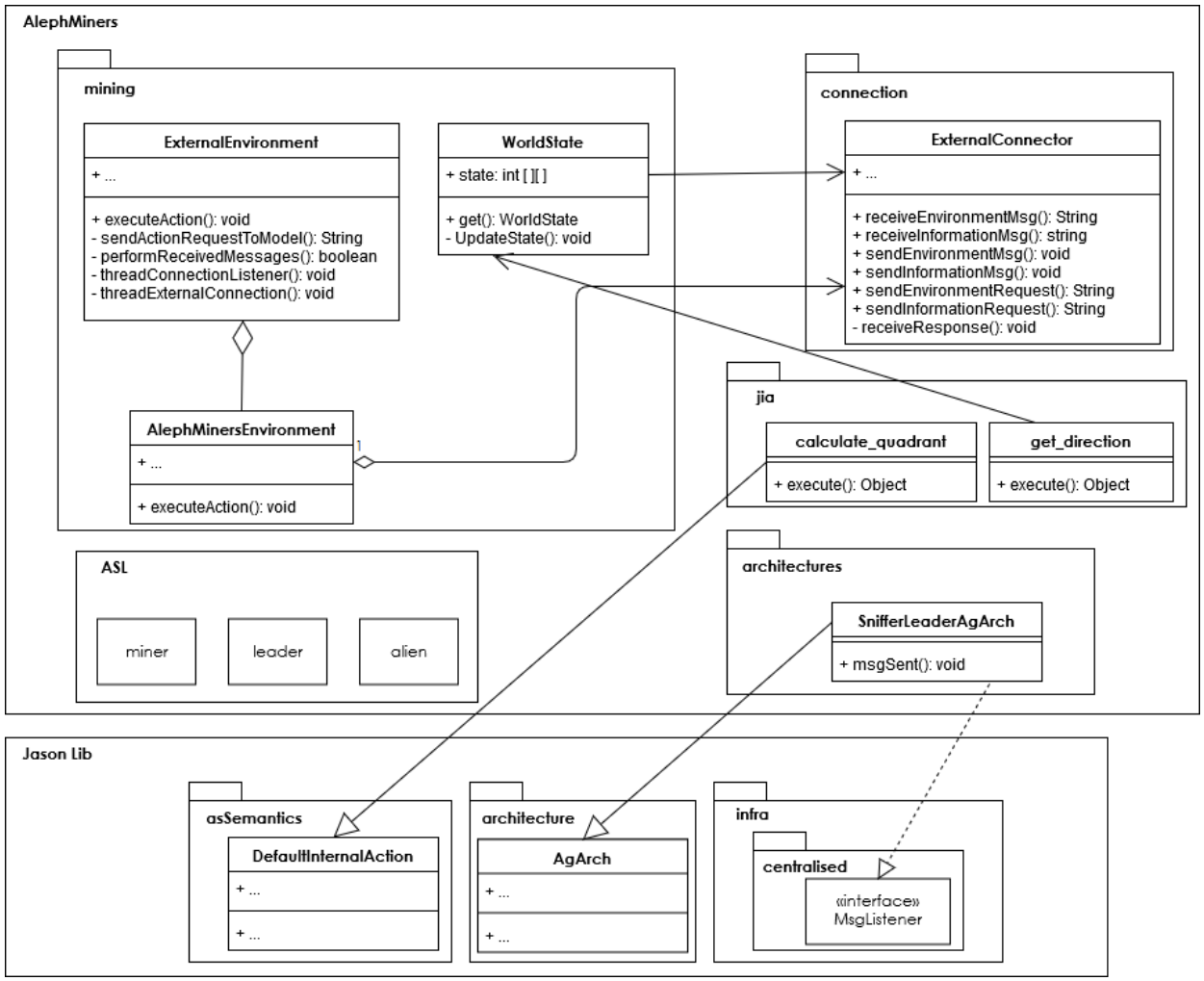

Figura 5.4: Diagrama de clases en Java

En el diagrama se omiten todas las clases que no han sido creadas por el autor de este trabajo, salvo aquellas que guardan una relación estrecha con las que sí son originales, como es el caso de *get-direction* y su relación con *Worldstate*. También se omiten las clases de *Connection*, ilustradas en el diagrama de *UniJason*. El marco "ASL" señala los ficheros de los tres tipos de agentes del sistema.

Diagrama de clases en Unity

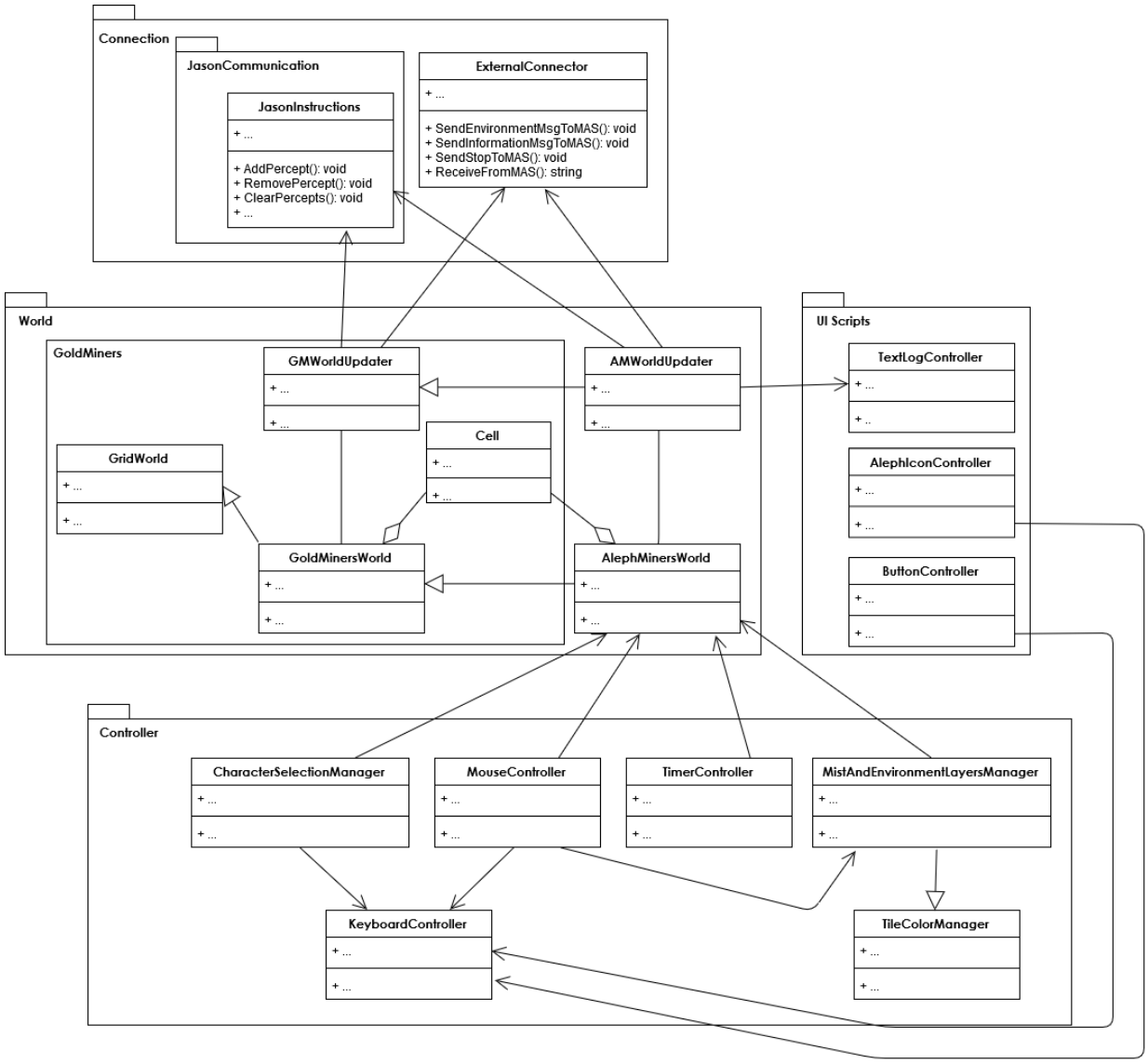

Figura 5.5: Diagrama de clases en Unity

El marco "GoldMiners" señala las clases que pertenecen a la versión de su adaptación a Unity. También vuelven a aparecer algunas clases de *UniJason*, a fin de enseñar de qué manera se unen las clases con la herramienta. Por supuesto, en el diagrama no aparecen las referencias que tienen algunas clases a objetos que aparecen en la escena de Unity.

5.2.3. Implementación

5.2.3.1. Versión 1: Adaptación de *GoldMiners*

En este apartado se explicará cómo se ha realizado la implementación a ambos extremos, primero en Unity y después en Java.

Implementación en Unity

El foco principal de la implementación de la adaptación se pone en la conexión del controlador con *UniJason* y la construcción de la vista. Lo relacionado con la implementación de *GoldMiners*, es decir, el modelo y el controlador, han sido traducidos directamente de Java a C# originando las clases del modelo *GridWorld*, de la cual extiende *GoldMinersWorld*, y *GMWorldUpdater* como controlador de ésta, tal y como se comenta en el análisis. Ambas clases están instanciadas en un *GameObject* vacío en el editor de Unity llamado *Board*, que representa la lógica del videojuego.

Controlador de la adaptación

Sin embargo, han sido necesarios ciertos añadidos propios a *GMWorldUpdater* para su unión con *UniJason*. La estructura de la clase se detalla a continuación. Se caracteriza principalmente por el método *CheckEvents*. Éste es ejecutado desde la función *Update* —ejecutada cada *frame*— de *GoldMinersWorld*. A él llega un *array* de *strings* con la cabecera de ámbito y el contenido del mensaje recibido a través del *ExternalConnector*.

Dependiendo del ámbito, ejecuta *MentalActionRequest* si el nombre del mensaje JSON es *stateRequest*, o *ActionRequest* si es *do*. Trataran el contenido del mensaje para los ámbitos de Información y Entorno, respectivamente. *MentalActionRequest* está implementado para responder un único tipo de petición: la matriz bidimensional de enteros contenida en *GoldMinersWorld*, con la que después trabajará la clase de Java *WorldState*, de la que se hablará más adelante. Se ejecuta *ToStringState*, que genera una cadena con la información de la matriz, la que es devuelta para ser mandada a Java a través de *SendResultMentalAction*.

ActionRequest identifica a través de la función *ParseAction* la acción requerida por el agente que la solicita, que realiza la llamada a la función correspondiente en *GoldMinersWorld*, la cual modifica la matriz de datos y la vista del juego. Después, el resultado de esa llamada —que es *true* si la acción se ha realizado, o *false* si no— se utiliza en *SendSuccededAction* para informar a Java de dicho resultado en forma de mensaje JSON a través del *ExternalConnector*. Por último, se actualizan las percepciones del agente que ha ejecutado la acción vía *UpdateAgPercepts*. Por lo demás, *GMWorldUpdater* contiene las mismas funciones que el controlador original de Java.

Modelo de la adaptación

Con *GoldMinersWorld*, como se ha dicho, se hizo una traducción directa, a la que se le sumaron los métodos necesarios para elaborar y modificar la vista en la *scene* de Unity. Para ello, también se crearon clases adicionales que se comentan a continuación. Al inicializar la clase, se ejecuta una de las funciones *World#* de la clase a través de un *switch*, el cual está determinado por el valor de un registro entero accesible desde el inspector del *script*, en Unity. Cada función *World#* se compone de llamadas a funciones que inicializan el modelo y la vista. En primer lugar se encuentra la llamada al *Initialize* de *GridWorld*, que crea la matriz bidimensional y el *array* de clase *Location* que contendrá las posiciones de cada agente, siendo el tamaño de ambas estructuras el determinado por los parámetros de entrada.

Después se llama a *InstantiateCells*, que construye el mapa en la vista. Lo que hace es instanciar array bidimensional de celdas, *cellsBoard*, y a continuación recorrerlo, almacenando en cada posición un objeto de tipo *Cell* (que será una script de Unity) devuelto por la función *InstantiateCell*, a la que se le pasan los dos indices del bucle. InstanciateCell toma un objeto público, un *CellPrefab*, proporcionado desde el inspector de Unity para generar una nueva instancia de dicho objeto, un *GameObject*. A continuación, como posición de esta nueva instancia en el espacio 3D de la *scene* de Unity, se le asigna la posición dada por los índices pasados por parámetro a la función. Por último, se devuelve el objeto *Cell* contenido en dicho *GameObject*. Las siguientes instrucciones de *World#* son llamadas a *SetAgPos* y *Add*, para colocar a los agentes en posiciones concretas y añadir objetos como obstáculos o depósitos de oro, respectivamente.

Los métodos de *GridWorld*, *SetAgPos Add* y *Remove* son re-implementados para poder incluir las modificaciones de la vista. En el *SetAgPos* original se comprueba si el agente que se quiere insertar en una posición quiere moverse, si ya existe y por lo tanto parte de una posición origen, o quiere crearse en ella, si por el contrario no existe aún. De esta forma sabe si eliminar o no la información relativa al agente en la posición origen. Y de la misma forma se hace en la función de *GoldMinersWorld*: primero se ejecuta el *SetAgPos* original, y después, si el agente ya existe en alguna posición, llama a *RemoveFromView*, pasando por parámetro el código entero que identifica al objeto a eliminar y la posición concreta del *Cell* dentro de *cellsBoard*. E independientemente de si ya existía o no, llama a *PlaceOnView*. *Add* y *Remove* ejecutan también los originales de *GridWorld*, ejecutando *PlaceOnView* o *RemoveFromView*, respectivamente.

PlaceOnView y *RemoveFromView* envuelven a las funciones *PlaceHere* y *RemoveFromHere* de la *Cell*, contenida en cellBoard, especificada por la posición pasada por parámetros. De éste modo, sólo queda por explicar la implementación de la clase *Cell*, sus funciones *PlaceHere* y *RemoveFromHere*, y su relación con la vista. La clase *Cell* describe la manera en que los todos objetos que pueden aparecer en el mapa se sitúan en cada *Cell*. Por este motivo, cada *Cell* posee una variable pública de tipo *GameObject* por cada tipo de objeto (Agentes, obstaculos o depósitos de oro) que puede haber sobre ella. Y es desde el inspector de Unity se asigna a cada variable pública el *prefab* correspondiente a dicha variable. Al inicializar el objeto *Cell*, se crea una instancia *GameObject* de cada *prefab* referenciado desde el inspector, y son desactivados. La activación de un objeto repercute en si es o no interactuable desde el entorno de juego. Desactivar un *GameObject* implica que éste pasa a no ser tangible desde el entorno de juego, pero sigue existiendo para el motor. Activarlo revierte el proceso. Por defecto, todos los objetos nuevos están activados.

Ahora es cuando entran en juego las funciones *PlaceHere* y *RemoveFromHere*. En lo que consiste éstas es en la activación o desactivación del objeto que se indica por parámetro, respectivamente. De esta manera, si se ejecuta *SetAgPos*, se llama al RemoveFromHere de la *Cell* origen, desactivando el *GameObject* agente —sólo en caso de que exista—, y el *PlaceHere* de la *Cell* destino activará la instancia del *GameObject* agente que hay almacenada en ella, haciendo visible al agente en el mapa. Esto crea una ilusión de movimiento para los agentes, sin que realmente se estén moviendo por el mapa: Lo único que hacen es desaparecer y reaparecer en la casilla adyacente, lo cual es una solución bastante cercana a la forma en que se simula el movimiento en la versión original de *GoldMiners*.

Implementación en Java

En el extremo de Java, hubo que realizar una clase *WorldState* a la que accedería la acción interna *Get-direction*, dado el traslado del modelo a Unity. Originalmente, *Get-direction* coge una referencia a la instancia del modelo con el método *Get* de *WorldModel*, y realiza el algoritmo A*, preguntando primero al modelo si la casilla destino está libre de obstáculos por medio del método *IsFreeOfObstacle*. En WorldState lo que se ha implementado una matriz bidimensional que contiene los datos de la matriz bidimensional de Unity, y una versión adaptada de los métodos *Get* e *IsFreeOfObstacle*. Cuando *Get-direction* llame a *Get*, crea la instancia si no existe y llama a *UpdateState*. Éste solicita a Unity una copia del estado actual de la matriz de datos, utilizando el nombre JSON *stateRequest*, que se comentó al hablar del procesado de mensajes en Unity. Al recibir el mensaje que la contiene, ejecuta *UpdateStateMatrix*, que lo procesa y recorre la matriz local actualizando su información. Una vez hecho esto, la clase está actualizada para poder ofrecer información considerablemente fiable cuando se ejecute su método IsFreeOfObstacle. La información nunca puede ser totalmente fiable, ya que en el tiempo que se tarda en recibir la matriz, procesarla y acceder a su información, esa información ya ha podido quedar ligeramente desfasada en el modelo en Unity.

5.2.3.2. Versión 2: *MotBP* - Juego y Sistema de Clases

El presente apartado detalla en orden la implementación en Unity, Java y Jason (ASL). Para la implementación de la versión más básica del juego fue necesario implementar el sistema de clases, de ahí que se expliquen juntas. A continuación, se sigue la misma estructura que en el análisis del *GDD* para hablar de la implementación del prototipo. Como último apunte, aclarar que ahora hay dos tipos de agentes, los mineros y los enemigos.

Implementación en Unity

Modelo del videojuego

En *AlephMinersWorld* se añadió un array de clase *Location* (*enPos*), para las posiciones de los enemigos y análoga de la existente *agPos*, otro array con las clases de cada agente (*agClasses*), y arrays de modelos 3D: Uno con los modelos de cada clase(*classModels*); otro de agentes (*agModels*), que contiene cinco instancias de modelos contenidos en *classModels*, según la clase determinada en *agClasses*; y otro de enemigos (*enModels*), que como todos los enemigos son iguales, se compone de instancias del modelo almacenado en *enemyModel*. Todos estos arrays de modelos 3D son públicos, pero sólo *classModels* y *enemyModel* son accesibles y toman su contenido desde el inspector de Unity.

Como se menciona en el análisis, *AlephMinersWorld* extiende a *GoldMinersWorld*, aunque gran parte de sus métodos han sido re-implementados. A grandes rasgos, se ha retocado la forma en que se representa el movimiento de los agentes, haciendo necesaria la re-implementación de todos los métodos vinculados con él, entre ellos *Move* y *SetAgPos*, además de la adición de nuevos métodos relacionados con este propósito. Aunque *Move* es exactamente igual que en *GoldMinersWorld*, ha sido necesaria su re-implementación para evitar que se utilice el *SetAgPos* de *GoldMinersWorld*, ya que la función *SetAgPos* de *AlephMiners* sí que es distinta. *SetAgPos* ahora llama a *MoveOnView* en caso de que se ejecute para un agente existente.

MoveOnView consiste en una función que extrae la posición origen de aquel agente que quiere moverse, sea minero o enemigo, y ejecuta una corrutina, llamada *Movement*. Ésta corrutina se encarga de activar las animaciones de movimiento del modelo del agente que se quiere mover, y calcula, para un periodo X (actualmente medio segundo), cuánto se tiene que mover cada fracción *delta* de tiempo, y realiza cada movimiento hasta que llega a su destino o el tiempo es excedido. Se han añadido los métodos *MoveEnemy* y *SetEnPos*, que se comportan de forma similar a sus análogas para los mineros, accediendo, en lugar de a *agPos*, a *enPos*. Sin embargo, *Move* se diferencia de *MoveEnemy* en las comprobaciones que hace: *Move* verifica que la posición origen no esté ocupada ni por otro minero, ni por un obstáculo. *MoveEnemy* comprueba que no esté ocupada ni por otro enemigo, ni por un obstáculo. En el caso de *SetEnPos*, el comportamiento es completamente análogo.

También se ha añadido la función *Kill*, utilizable tanto por mineros como enemigos. Esta función recibe como parámetros qué tipo de agente la solicita, el id de dicho agente, y el id de su objetivo. Sólo se realiza si la posición del minero, procedente de *agPos*, y la del enemigo, de *enPos*, coinciden. Si es el enemigo el que realiza la acción, se comprueba si el minero objetivo es de clase Soldado. Si no lo es, se borra al agente de la matriz de datos y se elimina su posición de *agPos*. Si es un minero el que quiere realizar la acción, si es de clase Soldado, se procede de igual forma con los datos del enemigo.

InstantiateCells ahora añade la referencia de cada *Cell* instanciada en *InstantiateCell* dentro de cada *cellObject* generado (que se materializa en las *tiles* del tablero de la *scene*) en *tilesInQuad*, para poder ser utilizado posteriormente por el controlador de colores e iconos. *InstantiateCell* asigna a cada instancia el cuadrante al que pertenece y su posición en los ejes de coordenadas usando las funciones *SetQuad* y *SetPosition* de dicha instancia. También se han eliminado la mayoría de métodos *World*, dejando sólo una revisión del *World3* de *GoldMiners*, ya que en el prototipo solo tratamos con un mapa. Adicionalmente, se añadieron métodos para identificar a agentes conociendo su identificador, su posición, etc.

Controlador del videojuego

En *AMWorldUpdater* se re-implementó *CheckEvents*, añadiendo un nuevo nombre de mensaje procesable, *msgReceived*, al ámbito *Information*. Esta nueva opción posibilita la ejecución de *ChangeDisplayedEnvironmentInfo* cuya utilidad se comentará más adelante. *ParseAction* se amplía para cubrir las nuevas acciones *MoveEnemy* y *Kill*.

Por primera vez, se ha modificado la actualización de las percepciones de los agentes, añadiendo un procedimiento específico para los enemigos (*UpdateEnemyPercept*) análogo a *UpdateAgPercepts*, que también ha sufrido modificaciones. *UpdateEnemyPercept* se comporta como el *UpdateAgPercept* (la versión del procedimiento que toma el identificador del agente por parámetro) original de *GoldMiners*, con la salvedad de que no envía un percept de si que carga con oro (*carrying-gold*), ya que los enemigos sólo pueden moverse y matar. *UpdateAgPercept* ahora también inserta la percepción de la clase a la que pertenece el minero, y después de insertar la percepción de lo que hay a una casilla de distancia, si su clase es o Soldado o Explorador, ejecuta *UpdateAgIncreasedPercepts*, que amplía el rango de las percepciones insertadas a dos casillas de distancia. Para ambas versiones se ha añadido una comprobación al principio de cada método: Si al extraer el objeto Location de la posición indicada por el número de agente de *agPos*/*enPos* éste es *null*, no se procede con la inserción de percepciones corriente, sino que se inserta la creencia *dead*, para la cual el agente, ségun se describe en su código ASL, se autodestruirá. Esto se desarrollará más adelante.

ActionRequest también ha visto cambiada su estructura. En esta versión y debido a fallos a la hora de trabajar de los agentes con información desfasada, ahora antes de responder a la acción solicitada se actualizan sus percepciones vía *UpdateAgPercept / UpdateEnemyPercept*, y no después, como sí se hace en el código original de *GoldMiners*. Esta solución es la mejor sin ninguna duda, ya que el agente estará bloqueado hasta recibir una respuesta, consiguiendo que para cuando la reciba, se disponga de toda la información actualizada. Este cambio se vió necesario al ver que, cuando un agente evalúa si existen creencias, que son consecuencia de una acción, inmediatamente después de recibir la confirmación de dicha acción, puede no disponer de la información actualizada y conducir a error o a comportamientos inesperados. De esta forma se asegura la correcta disposición de información en cada momento.

Controladores

A continuación se detalla la implementación de los controladores. Todos los *scripts* que constituyen realmente a cada controlador se encuentran instanciados en *GameObjects* vacíos que representan de forma tangible para el editor de Unity a dichos controladores, con el objetivo de poder tomar las referencias de las instancias de *AlephMinersWorld* y *AMWorldUpdater* contenidas en *Board*, y las de los controladores en dichos *GameObjects* vacíos. Estos *GameObjects* están organizados, dentro de la jerarquía de la *scene* de Unity, en *Input-Output Controllers*.

Controlador de teclado

El controlador de teclado está representado en la jerarquía del editor por el *GameObject Keyboard Controller*, que contiene a su *script* homónimo. La clase *KeyboardController* se encarga de comprobar cada *frame*, a través de su *Update*, si el jugador pulsa una de las teclas entre el 1 y el 5, o la 0 del teclado normal (no numérico), se guarda en una variable entera privada (*pressedKey*) el número de la tecla pulsada. Adicionalmente, hay métodos *getter* y *setter* para acceder a esa variable desde fuera.

Para la activación y desactivación de la capa de casillas que oculta el mapa real al jugador se emplea la tecla 0, que no cambia el valor de *pressedKey*. Su gestión se lleva a cabo por *SwitchMistLayerState*, que introduce un retardo al pulsar la tecla para evitar que la capa se active o desactive más de una vez por pulsación. La clase guarda una referencia a la instancia del script contenida en *MistLayerManager*, asignada desde el inspector.

Controlador de selección de personajes

Al controlador de selección de personajes lo representa *Character Selection Manager*, que contiene el *script CharacterSelectionManagement*. Contiene referencias a las instancias de *AlephMinersWorld* de *Board* y de *KeyboardController*. También contiene cinco referencias publicas a *GameObjects*, referenciando los retratos de los personajes, colocados en el *canvas* de Unity. Por último, almacena la última tecla pulsada en *lastKey*. En el procedimiento *Start* de la clase se asigna cada retrato a la posición que correspondiente a cada personaje en *portraitsArray*, y los desactiva. Su *Update* llama a *UpdateSelectedCharacter*, que accede a la *pressedKey* del *KeyboardController* mediante *GetPressedKey* y llama a *FlickeringLight*. Después, si *lastKey* es distinta de la actual *pressedKey* del *KeyboardController*, quiere decir que se ha cambiado de personaje, por lo tanto ejecuta *SwitchCharPicture* y *SwitchCamera* y actualiza *lastKey*. Ambas funciones *Switch* desactivan todos los retratos/cámaras y activan sólo la correspondiente a la tecla pulsada.

Construcción de la pantalla para las cámaras en la escena La elaboración de la pantalla en la interfaz del juego que muestra las cámaras de los personajes no es trivial. Para construirla ha sido necesario crear un *RenderTexture*, *CameraRenderTexture*, que se asocia a una *RawImage* en el canvas, contenida en el *GameObject CameraDisplay*, que hace las veces de pantalla para las cámaras. Cada cámara tendrá asociado desde el inspector, en su campo *Target Texture*, la *RenderTexture CameraRenderTexture*, de forma que la imagen de cada cámara se vuelca en esa textura, que a su vez está aplicada sobre la componente *RawImage* de *CameraDisplay*. Por defecto, todas las cámaras estan volcando su imagen en *CameraDisplay*, pero al desactivar todas las cámaras salvo una, se vuelca únicamente la imagen de la cámara que permanece activa.

FlickeringLight se encarga de controlar el parpadeo en la luz del personaje seleccionado, alternando entre el color original de la luz y el color blanco cada diez *frames*, y restaurando el color original en caso de que se haya seleccionado otro personaje. De este modo, mientras *lastkey* sea igual a *pressedKey*, se lleva la cuenta de los frames que la luz ha estado de un determinado color, alternándolo cuando han pasado varios frames. En el momento que dejan de serlo, se restablece el color original de la luz del ultimo personaje seleccionado, evitando que ésta quede de color blanco, y se tiñe de blanco la luz del personaje seleccionado actualmente.

Controlador de coloreado de tablero, e iconos

El controlador de coloreado de tablero e iconos está representado por *Layers Tile Color Manager*, que instancia la clase *MistAndEnvLayersController*, la cual extiende a *TileColor-Management*. *TileColorManagement* fue pensada para reflejar todos los cambios producidos en el juego directamente sobre el tablero, cambiando el color de cuadrantes y casillas, y que estos cambios fueran visibles para el jugador. Posterioremente, al añadir la *Mist Layer*, se extendió la clase para poder reflejar en la *Mist Layer* los mismos cambios producidos en el mapa, y situando sobre esta capa, para las casillas que contienen Aleph, iconos interactuables con el ratón.

Probablemente *TileColorManagement* y *MistAndEnvLayersController* sean las clases más complejas de explicar, así que se les dedica bastante texto a continuación. *TileColorManagement* se compone de cuatro referencias públicas a *Materials* que suponen los 4 materiales con los que se pueden colorear las casillas del tablero: *defaultMat*, *markedMat*, *goldMat* y *dangerMat*. También almacena una matriz bidimensional de *Material*, *tilesMaterialMatrix*, de dimensión 9 x 144, siendo cada una el número de cuadrantes y el número de casillas en cada cuadrante. Por último, también contiene un array de *Material* de longitud 9, *quadMaterialArray*, que preserva el color predominante de cada cuadrante.

En su Start, se inicializan las estructuras y se ejecutan *InitQuadMaterialArray*, que recorre *quadMaterialArray* asignando el *Material defaultMat* a todas las posiciones; e *InitTilesMaterialMatrix*, que recorre todo el tablero almacenando los materiales iniciales de cada casilla, ya que no siempre son *defaultMat*, como es el caso de la casilla del depósito de Aleph, que es de color rojo desde la creación del entorno en los métodos *World* de *AlephMinersWorld*/*GoldMinersWorld*. Hay tres bloques de procedimientos principales: los que gestionan los cambios de color de casillas específicas, los que gestionan el marcado de cuadrantes, y los que gestionan los cambios de color de cuadrantes.

El primer bloque lo componen *SetTileGoldMaterial*, *RestoreTileMaterial* y *SetTileMaterial*. Las dos primeras llaman a la tercera con el *Material* apropiado: En *SetTileGoldMaterial* se llama a *SetTileMaterial* pasandole la variable *Material goldMat*, y *RestoreTileMaterial* hace lo mismo con el color predominante del cuadrante en el que se encuentra dicha casilla —calculado con *GetQuad*, función de *AlephMinersWorld*— extraído de *quadMaterialArray*.

El segundo bloque lo componen *SetMarkedMaterial* y *RestoreMarkedMaterial*. Se recuerda que el marcado de cuadrantes es el cambio de color que sufren los cuadrantes del tablero, resaltándolos, para dar *feedback* visual al jugador de que puede seleccionar un cuadrante concreto. *SetMarkedMaterial* recorre todas las casillas de un cuadrante concreto, cambiando su *Material* a *markedMat* sólo si el material de esa casilla, en *tilesMaterialMatrix*, coincide con el del cuadrante que contiene a la casilla en *quadMaterialArray*. Esto se hace así para no tapar el material de las casillas especiales y dejarlas visibles aunque el jugador resalte el cuadrante que las contiene. *RestoreMarkedMaterial* vuelve a colorear las casillas de un cuadrante con la información del material de cada casilla almacenada en *tileMaterialMatrix*.

El tercer y último bloque está compuesto por *SetDefaultMaterial*, *SetGoldMaterial*, *SetDangerMaterial* y *SetMaterial*. Las tres primeras llaman a la cuarta pasandole por parámetro sus respectivos materiales. *SetMaterial* recorre el cuadrante realizando la misma comprobación para cada casilla que *SetMarkedMaterial*, para no colorear casillas especiales. Pero en esta ocasión, si el material de la casilla es igual al del cuadrante, no sólo se colorea la casilla en el tablero, sino que se sobrescribe el material almacenado en *tilesMaterialMatrix* para la casilla asociada con el material pasado por parámetro. Al acabar de recorrer todo el cuadrante, también se guarda el material pasado por parámetro en *quadMaterialArray* para la posición correspondiente. De esta forma, cuando se marque un cuadrante, se restaurará con el color actualizado y no con el material por defecto, puesto que el cuadrante puede estar coloreado con el material *dangerMat*, avisando de la presencia de enemigos en ese cuadrante, o de *goldMat*, avisando de la presencia de Aleph. Todos los métodos que acceden a una casilla del tablero para cambiar su material lo hacen a través de *SetEnvironmentLayerTileChanges*, que puede cambiar el material de la casillas almacenadas en *tilesInQuad* en *AlephMinersWorld*, que guarda referencias a cada casilla del tablero de la *scene*.

Al extender la clase, en *MistAndEnvLayersController*, se hace necesaria la re-implementación de todos los métodos. Se añaden dos estructuras, *tilesInMistLayer*, que almacena las referencias a las *tiles* de la scene de la *MistLayer* como hace *tilesMaterialMatrix* con las del tablero; y *popUpsListArray*, que tiene la misma estructura que *tilesInMistLayer* y *tilesMaterialMatrix*, almacenando para cada casilla una instancia de *AlephButtonPrefab*. Éste *prefab* es asignado a la variable pública *GameObject alephIconPrefab* desde el inspector de Unity. Explicar la re-implementación de los métodos de *tilesMaterialMatrix* es sencillo: Cada vez que se invoca a *SetEnvironmentLayerTileChanges* para cambiar el material de una casilla, justo después se ejecuta un nuevo procedimiento, *SetMistLayerTileChanges*, que hace lo propio con las *tiles* de la *Mist Layer*.

Entre los nuevos métodos nuevos están *InitTilesInQuadMist* y *InitPopUpsListArray*, que inicializan ambas nuevas estructuras; *GetTileCopiesFromScene*, que mete en *tilesInMistLayer* las referencias de cada tile de la *Mist Layer* en la *scene* en su correspondiente posición en la estructura; y los que gestionan la aparición y desaparición de los iconos interactuables generados sobre las casillas donde se notifica que hay Aleph, *SetGoldPopUp* y *RemoveGoldPopUp*. Éstos son utilizados desde la re-implementación de *SetTileGoldMaterial* y *RestoreTileMaterial*, respectivamente.

SetGoldPopUp crea una instancia (*popUpIcon*) de *alephIconPrefab*, que da lugar a un icono en la *scene*. Esta instancia posee un componente *script AlephIconController*, el cual almacena las coordenadas X e Y recibidas por parámetros. Estas coordenadas son fijadas en el script del icono después de instanciar el prefab mediante el método *SetPoint*. Después, se alinea la posición del *popUpIcon* en la *scene* con el centro de la casilla sobre la que se debe situar, y ajusta su altura para sobresalir por encima de la *Mist layer*. Por ultimo se guarda la referencia a dicha instancia en la *popUpsListArray*, para poder ser accedida cuando sea necesario.

RemoveGoldPopUp también recibe por parámetro las coordenadas X e Y. Su funcionamiento se reduce a buscar en *popUpsListArray* una instancia de *alephIconPrefab* que, al ejecutar el método *IsOnPoint* de su script, para X e Y, devuelva *true*. Cuando esto sucede, se borra la referencia a la instancia en *popUpsListArray* y se destruye la propia instancia a todos los efectos.

Controlador de ratón

Al controlador del ratón lo representa el *MouseController*, que instancia la clase homónima. Ésta contiene referencias a las instancias de *AlephMinersWorld*, *AMWUpdater*, *MistAndEnvLayersController* y *KeyboardController*; referencias a cuatro *Texture2D*: *defaultTexture*, *alephTexture* y *quadrantsTexture*; y variables que controlan el flujo de ejecución. El método *Start* de la clase cambia la imagen del cursor del sistema operativo por *defaultTexture*. *Update* comprueba cada *frame* la posición del cursor, distinguiendo si está situado sobre un icono o sobre un cuadrante gracías a *Raycast*, de la clase *Physics* de Unity.

Raycast lanza un rayo invisible desde una posición origen y en una dirección, y devuelve si ha impactado contra un objeto. Se puede sacar como parámetro de salida su *RaycastHit* asociado, que contiene la información acerca del objeto con el que ha impactado. Adicionalmente, se puede llamar a *Raycast* con una máscara de capas, compuesta de bits, que determina con qué capas definidas en Unity puede impactar el rayo disparado. En cualquier caso, es imprescindible que un *GameObject* tenga una malla de colisiones (un componente *Mesh Collider*) para que el rayo pueda impactar contra él.

De este modo, como el rayo puede impactar en dos capas de interés para el juego, como son los cuadrantes y los iconos, hay que definir dos nuevas capas en Unity y sus máscaras correspondientes: *Tiles* y *Aleph Icons* como capas en el editor, y *tileLayerMask* y *AlephIconsMask* en la clase del controlador. También hay etiquetar al *TilePrefab* (cuyas instancias componen el tablero) dentro de la capa *Tiles*, y lo mismo con el *AlephIconPrefab* dentro de *Aleph Icons*, aparte de añadirles un componente *MeshCollider*.

Volviendo a *Update*, primero se llama a *AlephIconsLayerRaycastHitter*. Ésta función lanza un Raycast desde la posición del cursor en la pantalla. Si el rayo impacta contra algún objeto perteneciente a la capa *AlephIconsMask*, *RaycastHit.transform* contendrá toda la información del *GameObject* impactado —siendo éste una instancia de *AlephIconPrefab*—, pudiendo acceder a sus componentes. De esta manera, se llama al método *GetPressedKey* del controlador de teclado, y se verifica si la tecla pulsada se corresponde con la de un personaje de clase recolector. Si es así, se ejecuta *SendToAlephClickChecker*, pasando el RaycastHit por parámetro, devuelve *true* a *Update*.

SendToAlephClickChecker chequea si se ha presionado el botón izquierdo del ratón estando el cursor situado sobre un icono de Aleph para ejecutar el procedimiento *SendPickAlephOrder* del componente *AlephIconController* de la instancia de *AlephIconPrefab*, accedido a través *RaycastHit.transform*. Por último, las variables globales *CheckedClick* y *DelayClick* se actualizan a *true* y *30* respectivamente. La existencia de *DelayClick* se justifica en la necesidad de evitar que un jugador que ha clicado sobre un icono de Aleph lo haga de nuevo accidentalmente sobre un cuadrante, desviando al agente de su objetivo.

De nuevo en *Update*, si *AlephIconsLayerRaycastHitter* ha devuelto *true*, se cambia el diseño del cursor a *alephTexture*. Si por el contrario devuelve *false*, se llama a *TileLayerRaycastHitter* si el *DelayClick* es 0. Esta función realiza lo mismo que *AlephIconsLayerRaycastHitter* para la capa *Tiles*. Si no impacta contra un objeto de dicha capa —presumiblemente una casilla del tablero—, acaba la ejecución y devuelve *false*. Si lo hace, accede a su componente de instancia de *Cell* y ejecuta su función *GetQuad*, que devuelve el cuadrante al que pertenece, almacenandola en *hittedQuad*. Después, se comprueba que el valor de *hittedQuad* sea distinto del de la variable global *lastQuad*, que almacena el número del último cuadrante impactado, evitando colorear el cuadrante una vez por *frame*. Si lo es, en caso de que hubiera un cuadrante ya marcado se ejecuta el *RestoreMarkedMaterial* de la instancia de *MistAndEnvLayersController*, y tanto si lo había como si no, se ejecuta el método *SetMarkedMaterial* de la misma instancia. Finalmente se actualiza *lastQuad* con el valor de *hittedQuad*, y *TileLayerRaycastHitter*devuelve *true*.

Si *TileLayerRaycastHitter* devuelve *true*, se cambia la textura del cursor a la almacenada en *quadrantsTexture*, y se ejecuta *SetNewQuadrantClickChecker*. Si no, se ejecuta *UnmarkLastQuadrant*. *SetNewQuadrantClickChecker* comprueba que el botón izquierdo del teclado haya sido pulsado. Si es así, se accede a la tecla pulsada a través de *GetPressedKey* del *KeyboardController*, y su valor está entre 0 y 1, envía a Jason un mensaje *AddPerpect* (*set-new-quadrant*), con el cuadrante resaltado, al minero que se corresponde con la tecla pulsada por teclado.

UnmarkLastQuadrant sólo se ejecuta cuando no se ha detectado el cursor encima del tablero, por lo que llama a *RestoreMarkedMaterial* para *lastQuad*, y actualiza *lastQuad* a *-1*. De esta manera, cuando el cursor no está situado sobre el tablero, ningún cuadrante queda marcado, y como *TileLayerRaycastHitter* sólo restaura el color en caso de haber un cuadrante ya marcado (*lastQuad != 1*), sólo colorea el nuevo cuadrante sobre el que se situa el ratón después de haberlo sacado del tablero.

Controladores de botones

Resta por contar la implementación de la lista interactiva para seleccionar a los personajes, la del temporizador, y la del *log* con los mensajes de los agentes. Esa lista interactiva se ha implementado haciendo uso de los botones para interfaz proporcionados por Unity, a los que se les puede asociar un script y determinar qué procedimiento de ese script se desea ejecutar al pulsar el botón. De esta forma, se crearon cinco botones apilados en la interfaz con los nombres de cada personaje, cuyo script asociado es uno muy simple llamado *ButtonController*. Esta clase contiene una referencia a la instancia de *KeyboardController*, y un único método, *SelectCharacter*, que recibe un entero por parámetro y llama al método SetPressedKey del *KeyboardController* pasándole ese mismo entero, actualizando su variable *pressedKey* en función del valor de éste.

Controlador del contador

Para el contador regresivo se ha creado un *GameObject* vacío, dentro del *Canvas* en la jerarquía de la scene de Unity, llamado *Time Field* que tiene como componentes un *Text* y el script *TimerController*. En el *Update* de esta clase se resta el tiempo que queda a través del método *TimeLeft*, comprobando después si se ha alcanzado el número de aleph necesario para ganar, parando el contador y mostrando un mensaje felicitando al jugador si se cumple. Si no se cumple, se comprueba si ya se ha acabado el tiempo, en cuyo caso se muestra un mensaje que informa al jugador de que ha perdido.

Controlador del *log*

Por último, la implementación del *log*. Para implementarlo, en el extremo de Unity, se necesita poder *parsear* la información procedente del sistema multi agente, y poder verla en una ventana con *scroll* vertical. El primer próposito lo realiza el procedimiento *ChangeDisplayedEnvironmentInfo* de *AMWorldUpdater* que no se detalló en su momento para hacerlo ahora. Esta función le pasa a *PerformChangesOnEnvironment* el cuerpo del mensaje JSON recibido, que produce cambios tanto en el *log* como en el tablero de juego de la scene. Esta función lo que hace es analizar el contenido del mensaje, dividiendo el flujo de análisis en cuatro bloques: Si el mensaje habla de un cuadrante con Aleph, si habla de un cuadrante con enemigos, si habla de una casilla con Aleph, o si no habla de ninguna de ellas. En los tres primeros bloques se producen cambios en el tablero de juego, y el último es el que los genera para el *log*. Para el primero se busca la palabra *"goldInQuadrant"*, analizando si va ligada a otra palabra clave, *"clear"*, en cuyo caso restaura el color del cuadrante al que se refiere el mensaje invocando al SetDefaultMaterial de *MistAnEnvLayerController*, y si no contiene *"clear"*, se llama al *SetGoldMaterial* del mismo controlador. Para el segundo bloque se hace lo mismo con *"alienInQuadrant"* y otra palabra clave *"killed"*, llamando a *SetDefaultMaterial* para el cuadrante especificado en el mensaje si aparece ésta última palabra, o a *SetDangerMaterial* si no lo hace. Para el tercer bloque lo mismo con *"gold"* y *"picked"*, invocando a *RestoreTileMaterial* si aparece la segunda palabra en el mensaje, o a *SetTileGoldMaterial* si no lo hace. En el último bloque se tratan todos los mensajes que no sirven para el coloreado del tablero, mostrándolos literalmente en el *log*. Aunque no es la implementación más sofisticada, abre las puertas a poder *parsear* según qué mensajes para darles forma y que no se muestren en formato mensaje ASL, puediendo descartar los que puedan resultar irrelevantes al jugador.

Construcción de la ventana de *log* en la escena

El *log* en la *scene* de Unity está implementado en el *GameObject Text Log Received*, situado dentro del *canvas* en la jerarquía. Éste tiene como componentes asociadas una imagen que hace de fondo y de un ScrollRect, un componente pensado para asociarle una *Scrollbar* en el que se configura cómo será el deslizamiento del texto: Vertical u horizontal, con o sin inercia, con o sin rebote, y se especifica el tamaño que debe tener en ejecución el *handle* del *scrollbar*. Text Log Received tiene *GameObjects* hijos en su jerarquía: *Viewport* y *Scrollbar*. *Viewport* tiene como hijo a su vez de un objeto *Content*, que tiene como componentes esenciales un *Vertical Layer Group*, que configura cómo se ajusta el texto en el deslizamiento vertical, y *Content Size Fitter,* que ajusta el tamaño del texto al del *Viewport*. Content también tiene otro hijo, *Text*, que se referenciará desde *AMWorldUpdater* por la variable pública *textLeaderReceivedMsgs* para volcar en ella todos los mensajes que llegan al cuarto bloque de *PerformChangesOnEnvironment* que se comentaba antes. *Scrollbar* tiene como componente un *ScrollBar* que configura el comportamiento de la barra de *scroll*, si va de abajo a arriba, si es interactuable... y qué objeto es el que actuará como *handle* del *scrollbar*. Como hijo de *Scrollbar* se encuentra *Sliding Area*, que sólo delimita el movimiento de su hijo, *Handle*. *Handle* se compone esencialmente de la imagen utilizada para hacer las veces de *handle*.

Con esto acaba la implementación en el extremo de Unity. A continuación se explicarán las implementaciones realizadas en Java, y los cambios realizados en el código ASL de los mineros respecto al original de *GoldMiners* .

Implementación en Java

En el extremo de Java, las implementaciones hechas en este lenguaje han consistido en la construcción de una nueva arquitectura para el agente líder, *SnifferLeaderAgArch*; el desarrollo de una clase que extiende a *ExternalEnvironment*, *AlephMinersEnvironment*; y la creación de una *InternalAction* para Jason, *Calculate-quadrant*.

Arquitectura de escucha de mensajes

Como se comentó en la parte de Unity, para poner mensajes en el *log* primero hay que capturarlos en el extremo de Java/Jason. Las primeras aproximaciones que se hicieron para resolver este problema proponían insertar en el código ASL del líder planes que utilizaran una *InternalAction* implementada en Java, que usaría el *ExternalConnector* de *UniJason* para enviar los mensajes escuchados por el líder. Esta opción resultó ser indeseable por dos razones: la primera, que el agente no debería encargarse de llevar a cabo tareas que no son de puro razonamiento, y la segunda y más importante, que para lograr esto desde el código ASL del líder habría que haber implementado llamadas a dicha InternalAction para cada posible mensaje escuchado por el líder, lo cual resta muchísima extensibilidad al desarrollo del código de los agentes que interactúan con él, como es el caso de los mineros.

De este modo, buscando una forma más eficaz de atajar el problema se consultó el manual oficial de Jason (Rafael H. Bordini, 2007) que habla de poder acceder a nivel arquitectónico a los mensajes que escucha un agente, extendiendo la clase *AgArch* que define la arquitectura más básica para los agentes, que es la usada por defecto para éstos si no se especifica lo contrario en el archivo *mas2j* que se utiliza para la ejecución del sistema multiagente.

Lo que no viene descrito en ningún lugar del manual es la existencia de un interfaz expresamente creado para escuchar mensajes: *MsgListener*. Al utilizar esta interfaz en una clase que extiende de *AgArch*, se necesita re-implementar el metodo *Init* de *AgArch*, para poder añadir la nueva arquitectura dentro de las arquitecturas capaces de escuchar mensajes en *CentralisedAgArch*.

Para la captura de mensajes propiamente dicha hay que implementar el único método que posee la interfaz, *MsgSent*, que se ejecuta para cada mensaje recibido, pasando por parámetro el mismo mensaje, de clase *Message*. Cabe mencionar que este método captura todo el tráfico, dejando a implementación el descartar o tratar los mensajes que se deseen.

Así, haciendo uso de *MsgListener* se ha construido *MySnifferLeaderAgArch*. En su método *MsgSent* lo que se hace es analizar si el destinatario de ese mensaje es el líder (*"leader"*). Si lo es, se construye una cadena que contiene quién ha mandado el mensaje y el contenido de dicho mensaje. Esta cadena se introduce en el campo *message* de un mensaje JSON cuyo nombre es *msgReceived*. Finalmente es enviado a Unity a través del *ExternalConnector* de *UniJason*. Debido a que es el jugador el que ahora asume el papel del líder en el videojuego, es el líder el agente que porta una arquitectura capaz de escuchar y enviar la información a Unity.

Extensión del entorno genérico de *UniJason*

La implementación de *AlephMinersEnvironment* consiste simplemente en añadir una array con la clase de cada personaje, y re-implementar unicamente el método *ExecuteAction*. Esta re-implementación consiste en añadir, justo antes de salir del método, esperas (*sleep*) de una duración distinta para cada clase de minero y para los enemigos.

De esta manera se cumple con la especificación de clase del *GDD*, en la que la clase Soldado es la más lenta, la clase Explorador es la más rápida y la clase Recolector está entre ambas. El retardo en los enemigos tiene que ser menor que en las demás clases para que puedan matar a los mineros.

Esta restricción se debe a la velocidad con que los agentes realizan acciones en el modelo: Si un enemigo quiere matar a un minero en su misma posición, pero para cuando su solicitud de acción es procesada en el modelo ese minero se ha movido de posición, la acción falla. Se podría pensar que ese mismo problema lo tendrían los soldados con los enemigos, pero a la inversa no funciona exactamente igual, debido a que la inteligencia de los enemigos está hecha para encarar a cualquier minero, y es cuestión de tiempo que un soldado consiga adelantarse a su enemigo al solicitar la acción y consiga matarlo antes de que éste cambie de posición, por ejemplo.

En cualquier caso, esta aproximación para implementar los retardos no es la deseable, ya que el contenido del array que almacena la clase de cada agente está implementado de forma explicita en *AlephMinersEnvironment*, lo cual funciona para el prototipo, pero debería ser rediseñado de cara al juego final.

Cálculo de cuadrantes usando Acciones Internas

A continuación se habla de la implementación de la *InternalAction Calculate-quadrant*. Calcula el cuadrante al que pertenece una posición X e Y, en función del numero de cuadrantes en que está dividido el mapa y de sus dimensiones. Esto sirve a los mineros para alertar de la presencia de enemigos, y a los soldados y recolectores para señarlar la presencia de Aleph, en cuadrantes concretos. Este cálculo se podría haber llevado a cabo en el modelo, haciendo que los mineros informen de posiciones exactas al líder, y éste al modelo, y en Unity se podrían esconder dichas posiciones al jugador coloreando cuadrantes enteros, en lugar de casillas concretas. Pero hacerlo de esta modo iría en contra de la coherencia de la inteligencia del sistema multi-agente, ya que en ese caso, ellos sí que poseerían la información exacta de las posiciones de un enemigo o veta de Aleph, cuando se supone que no todas las clases pueden. A efectos visibles no es algo importante, hasta que queremos utilizar esa misma información que posee el jugador para crear comportamientos verosimiles para los personajes. Por ello, se ha optado por sesgar la información directamente en el sistema multiagente, unificando lo que saben los agentes del entorno con lo que ve el jugador.

Implementación en ASL

Finalmente, es el momento de hablar de la implementación de los códigos ASL, tanto el del líder como el de los mineros. En las siguientes líneas se sintetizan los cambios producidos respecto al código original de *GoldMiners* para ambos códigos, con la mayor claridad posible.

Estos cambios tienen como objetivo incorporar un sistema de clases a los mineros, y borrar algunos comportamientos autónomos en los mineros, pero sobre todo en el líder, puesto que su papel ahora lo asume el jugador.

La implementación de los enemigos no se comenta, al ser una versión muy simplificada de la de los mineros.

Implementación del líder

El líder es quien presenta los cambios más rápidos de explicar. En el código original de *GoldMiners*, al comienzo de la simulación los mineros informaban al líder de su posición en el mapa, y éste asignaba cada cuadrante a cada uno de los cuatro mineros, en función de su proximidad. En el prototipo se ha optado por prescindir de esta asignación particular, asignando a todos los mineros el mismo cuadrante, el central, que representa la base de operaciones de los protagonistas del juego, y informándoles de que ése cuadrante representa la base. También, en el código original, los mineros pujaban por ir a por una pieza de oro encontrada con la distancia a la que se encontraban de ella, como moneda de cambio, asignando el líder al minero más cercano. Esto se elimina también, ya que la asignación de Aleph no está automatizada, sino que la realiza el jugador. En definitiva, lo que antes era el líder, ahora sólo almacena la información de las coordenadas de cada cuadrante, y asigna el cuadrante inicial e informa de que es la base, al principio de la simulación.

Implementación de los mineros

Para contar los cambios sobre los mineros, se hablará primero de los cambios realizados para la interacción con el jugador en Unity, y después se explicará los llevados a cabo para la implementación de las clases. Hay que destacar, para mejor comprensión de las siguientes líneas, que cuando un agente adquiere una nueva creencia, ya sea por inferencia, por su percepción o porque se lo ha comunicado otro agente, se dispara, si existe, un plan que se llama del mismo modo. Si existe un plan con el mismo nombre, a estas creencias se las llama *triggering conditions*. También hay que recordar que los planes se componen de dos partes: el contexto, que definirá si el plan es realizable o no, y el cuerpo del plan en sí, lo que se hace.

Planes para la interacción desde Unity

De este modo, para la interación con el jugador se han creado 4 planes que se disparan al percibir o ser informados de 3 creencias distintas: *dead*, *set-new-quadrant* y *go-to-pos*.

Cuando un agente recibe *dead*, consulta su propio nombre con la *InternalAction my-name(N)* y ejecuta otra *InternalAction* llamada *kill-agent(N)*, que elimina del sistema a aquel agente cuyo nombre unifique con N. Es decir, cuando un agente cree *dead*, se mata a sí mismo en el sistema, ya que, como se mencionó en la sección de Unity, *dead* se envía a un agente en Jason cuando ya no existe en el modelo, porque lo han matado.

Set-new-quadrant(Q) tiene dos planes para la misma *triggering condition*. Dado a que la evaluación se hace de arriba hacia abajo, en el contexto de la primera se comprueba si el agente es recolector y si está ocupado, es decir, si está llevando Aleph (es decir, si tiene la creencia de que lo está haciendo, *carrying-gold*) o si se dirige a una veta de de Aleph. Si es el caso, le envía un mensaje al líder de que está ocupado, y desecha la nueva creencia *Set-new-quadrant*.

Si no es el caso, salta al siguiente plan, que no tiene ningún contexto y se realiza siempre que la anterior no se ha realizado. Si se lleva a cabo este plan es porque el agente ni lleva Aleph ni va a por él, así que primero se olvida de cual es su cuadrante asignado mediante la *InternalAction abolish*. Desecha sus intenciones y eventos de moverse por un cuadrante, *around*, y de avanzar hacia una posición, *next-step*, con *drop-intention* y *drop-event* para cada intención/evento. Después se elimina *Set-new-quadrant* de la base de creencias, y añade Q, que será una creencia de la forma *quadrant(X1,Y1,X2,Y2)* y el nuevo cuadrante por el que tiene que moverse, y finalmente se añade *free* a la base de creencias. El motivo de esto es que *free* es otra *triggering condition* de un plan que hará, simplificándolo mucho, que el agente se dirija hacia el nuevo cuadrante.

Recordar que *go-to-pos* se manda desde Unity para dirigir a recolectores a posiciones donde hay Aleph. Para esta condición hay otros dos planes que hacen exactamente lo mismo. Si el agente es un recolector y está ocupado, informa al líder de que está ocupado y olvida *go-to-pos*. Si no lo está, desecha las mismas intenciones y eventos descritos para *set-new-quadrant(Q)*, olvidandose también de *go-to-pos* e informando al líder de que va a por el Aleph asignado. Finalmente ejecuta el plan *init-handle(gold(X,Y))* que prepara al recolector para recoger el Aleph, ejecutando posteriormente el plan *handle(gold(X,Y))*.

Planes para la recolección de Aleph

Respecto al sistema de clases, se han modificado los planes referentes a recoger oro, se han copiado y adaptado los planes para recoger oro para obtener los planes para matar enemigos, y se han añadido planes propios, uno explícito, *!return-to-base*, y el resto son *triggering conditions*.

Los planes relevantes son aquellos que tienen que ver con la detección y extracción de Aleph y con la detección y eliminación de enemigos, definiendo principalmente comportamientos distintos para cada clase para un misma *triggering condition*. Recordar que las creencias de estructura *cell(X,Y,Objeto)* se insertan desde el método *UpdateAgPercepts* en Unity, que manda a Jason tantos mensajes como casillas adyacentes al agente tengan objetos/entidades. De este modo, para cuando alguien detecta Aleph (*cell(X,Y,gold)*) hay configurados cuatro comportamientos (planes) para la misma condición.

El primer plan es para los recolectores. Si el agente es recolector, no lleva Aleph y está libre (*free*), entonces deja de estarlo, añade la posición del Aleph a su base de creencias e inicia el plan de cogerlo, *!init-handle*. Éste plan no cambia respecto al original, pero sí lo hace *!handle*, que es un plan que *!init-handle* acaba disparando. En el *!handle* original se comprobaba que el agente no estuviera libre, haciendo que éste anunciase que se dirige a por ese oro (*committed-to-gold*), fuese a por el oro objetivo con *!pos*, cogiéndolo con *ensure(pick, gold(X,Y))* y anunciando que lo ha recogido a los demás agentes una vez lo ha hecho. Después lleva el oro al depósito y lo deja con *ensure(drop,0)*, para finalmente considerar de qué otras piezas de oro conoce su posición, para ir hacia ellas, con *!choose-gold* .

Ahora el contexto es ligeramente distinto, y se añade el cálculo del cuadrante de la posición con la *InternalAction .calculate-quadrant* en función de la posición X Y del Aleph, y de *num-quads* y *gsize*, dos creencias globales insertadas en el inicio de la ejecución desde Unity. Esto es así puesto que ahora el recolector no sólo informa a todos los agentes de que ha recogido Aleph en una posición determinada (*picked(gold(X,Y)*), sino de que lo ha hecho en el cuadrante al que pertenece esa veta, con *picked(goldInQuadrant(Q))*. Esto es así para reestablecer tanto el color de la casilla como el del cuadrante, ya que el Aleph recogido puede haber originado antes la marca de todo el cuadrante o de sólo una casilla, dependiendo de quien la hubiera detectado.

Esta es la razón de que en el contexto se calcule el cuadrante al que pertenece la casilla con Aleph. Y se sustituye *!choose-gold*, que ya no existe puesto que estaba vinculada a la puja de distancias con el líder —y por lo tanto, con el automatismo de los agentes—, por *!return-to-base*. Este plan devuelve al agente al cuadrante central, haciendo que olvide su cuadrante actual (el que tuvieran asignado), y lo sustituya por el cuadrante base (el *base(Quadrant)* proporcionado por el líder), y se disponga libre, lo que, como se mencionó antes, conducirá a que los mineros se dirijan y caminen por ese cuadrante. En caso de que *!handle* fallase en algún punto, se ejecutarían los planes "de contingencia" (declarados como *-!handle* en este caso), haciendo que el recolector volviera a la base.

El segundo plan para *cell(X,Y,gold)* se ejecuta cuando un recolector se dirige a una posición con Aleph —ni está libre ni carga con oro, y que se encuentra siguiendo un plan *handle*— y encuentra una veta de Aleph que está más cerca que a la que se dirige. Se quita los deseos de ir a la veta más alejada de Aleph, compartiendo con todos los agentes que ha cambiado de objetivo, desdiciéndose en su *committed-to-gold*. Anuncia de nuevo la posición de ese Aleph del que se ha desentendido, e iniciando el plan *int-handle* para la nueva veta de Aleph.

El tercer plan para la *triggering condition cell(X,Y,gold)* es para los exploradores. Si el minero que adquiere esta creencia es explorador y no se ha notificado ya la existencia de oro en esa posición, guarda en su base de creencias la posición de ese oro y la comparte con todos los demás agentes con la *InternalAction .broadcast*.

El cuarto y último plan es para todos los demás, que en este caso son sólo los soldados. Si no hay constancia de oro en esa posición concreta, se calcula en el contexto el cuadrante al que pertenece dicha posición, como en *!handle*, valiéndose de *num-quads*, *gsize* y usando la *InternalAction Calculate-quadrant*. Entonces lo unico que hace es añadir a su base de creencias que hay oro en ese cuadrante, con *goldInQuadrant*, y compartirlo con el resto de agentes.

Planes para la eliminación de agentes

Para el caso de matar enemigos ocurre algo similar, pero con la *triggering condition cell(X,Y,alien)*. Esta vez también hay tres planes para ella, pero dos de éstas son para los soldados y una para el resto de clases. El primero en su contexto comprueba que el agente esté libre y sea soldado, y concreta las creencias necesarias para calcular el cuadrante con *.calculate-quadrant* como se ha visto hasta ahora. En el cuerpo del plan el minero deja de estar libre, añade a sus creencias la posición del enemigo (*alien(X,Y)*), y comunica a los demás agentes que ese cuadrante hay un enemigo (*.broadcast(alienInQuadrant(Q))*). Finalmente ejecuta *!init-kill* que se comporta se forma similar a *!init-handle*, desarrollando más adelante *!kill*, el análogo de *!handle*. El plan *kill* es exactamente igual que su análogo, pero con la salvedad de que el agente borra lo que sabe del *alien(X,Y)* de su base una vez lo elimina (ejecuta con éxito *!ensure(kill,alien(X,Y)*), lo que dispara una triggering condition para, *-alien(X,Y)*, que elimina con la *InternalAction abolish* cualquier recuerdo de posiciones de enemigos.

El segundo plan de *cell(X,Y,alien)* comprueba si el agente es soldado, está ocupado y tiene el deseo de matar a algun enemigo. Aún en el contexto, toma los valores AgX y AgY de *pos(AgX,AgY)*, que es la posición actual del agente, y compara la distancia entre el agente y el enemigo al que se dirige con la que hay entre él y el enemigo recién detectado. Si está más cerca el nuevo enemigo, se olvida de matar al anterior con *.drop-desire(kill(alien(oldX, oldY))* y ejecuta *!init-kill(alien(X,Y))* para el nuevo enemigo.

El tercer plan es el que desarrollan todas las clases que no son soldado, y al igual que pasa con la detección de Aleph para los soldados, simplemente se comunica con *.broadcast* la presencia de enemigos en el cuadrante.

5.2.3.3. Versión 3: *MotBP* - Sistema de Personalidades

Para las modificaciones de cada clase se han creado nuevos planes con mayor prioridad —esto es, que aparecen antes en el código— para condiciones disparadoras ya existentes. Todos ellos tienen una probabilidad mayor o menor de ser ejecutados por aquellos mineros que poseen determinadas personalidades.

Personalidad para soldados

Cobarde Cuando un soldado percibe un enemigo en sus proximidades, puede evitar matar a su enemigo. Primero, comunicaría que va a matar al enemigo y esperar un tiempo, simulando que está realizando la acción de matar. Después comunicaría a los demás agentes que ha matado al enemigo en ese cuadrante, e ignora al enemigo de ahora en adelante, para no avisar de su presencia (Como si ya no existiera).
Sin embargo, si otro minero avisa de la existencia de un enemigo en ese cuadrante —el que el soldado no ha matado y está ignorando— a éste no le queda remedio que dejar de ignorar al enemigo, y estar así disponible de nuevo para que el jugador lo envíe a matar al "nuevo" enemigo.

Disidente El soldado disidente siempre cumple con su cometido, matar, pero puede ignorar informar sobre presencia de vetas de Aleph en un cuadrante. De este modo, simplemente guarda en su base de creencias la localización de esa veta, de manera que, aunque la perciba de nuevo, no podrá anunciar que en ese cuadrante hay Aleph.

Personalidad para recolectores

Traidor Para esta combinación se contemplan dos casos, en función de si una determinada posición de Aleph es conocida o no por el jugador. Si no lo es, el recolector puede proponerse robar ese Aleph sin avisar a nadie —algo que no pasaría si no fuera traidor— de modo que el robo pasaría desapercibido para el jugador, y sigue haciendo lo que estuviera haciendo. Si la posición es conocida, se comporta de forma normal hasta que ejecuta la acción de recogerlo. Al ejecutarla puede decidir robarla, y avisar a todos de que en esa posición —ya— no hay nada (y también manda un mensaje para desactivar el icono en Unity) y vuelve a la base con normalidad.
Remarcar que esto es un comportamiento común a todos los recolectores. Si se creía que una posición contenía Aleph y se percibe que no lo contiene, el recolector avisa de la misma forma al jugador que si lo hubiera robado. Lo interesante es que esta comprobación la hace el traidor después de robar, reutilizando código existente.

Cobarde Siempre que un recolector sabe que en un cuadrante hay presencia enemiga, éste camina más despacio a través de él. De esta manera, se imita el comportamiento de la mente humana, actuando de forma poco racional ante el miedo.

Personalidad para exploradores

Disidente Este comportamiento es sencillo pero fatal para el jugador. El explorador disidente puede optar por no compartir una posición exacta de Aleph descubierta. Al igual que el soldado disidente, guardaría en su base de creencias la posición del Aleph percibido, de modo que, si lo vuelve a percibir, no dispare ninguna condición.

Traidor El traidor puede mentir al percibir presencia enemiga en el mapa. Esta mentira consiste en señalar como una posición con Aleph la posición donde se avista por primera vez a un enemigo en un cuadrante. De esta forma, recuerda dos cosas: Que hay un enemigo en ese cuadrante —no disparándose ninguna condición que compruebe el no conocimiento sobre enemigos en el cuadrante—, y que ha mentido sobre ese cuadrante, de manera que desde ese momento no mentirá más sobre él para evitar ser descubierto.

Capítulo 6

Pruebas y resultados

En este capítulo se reúnen las pruebas realizadas en la sesión de pruebas realizada que tenían como fin validar el diseño e implementación de la interfaz y lo intuitivo que resulta el juego.

6.1. Diseño de la prueba y método de evaluación

El propósito de la prueba es el de validar el prototipo como juego, evaluando la claridad de la interfaz y la distribución de sus elementos, y la complejidad de las mecánicas del prototipo. De esta manera se obtendría un *feedback* valioso que daría forma al que sería el videojuego final, puliendo los aspectos que se considerasen oportunos. El aspecto del prototipo final es el mostrado en la Figura 6.1.

Para ello, se consideraron ajustes sobre el prototipo. Para dificultar que el jugador pudiera ganar la partida, se marcó como meta en la partida conseguir 5 de 11 menas de Aleph repartidas por el mapa en un periodo máximo de 2 minutos (valores obtenidos experimentalmente en fase alpha). Debido a ello, el jugador sólo podría ganar si entendiese totalmente el funcionamiento de la demo.

Figura 6.1: Imagen del prototipo sin capa de niebla

Además, se creyó conveniente activar la *Mist Layer*, gracias a la cual se oculta todo el mapa salvo las posiciones de cada minero, identificados sobre la capa por una luz. Así, los sujetos de la prueba no sabrían la posición de los enemigos, que sí se muestran sobre el mapa real, ni tampoco la de los obstáculos. El resultado en pantalla es el mostrado en la Figura 6.2.

La decisión de ocultar los obstáculos se sustentó en el interés por ver qué interpretaciones hacía el jugador acerca de la inteligencia de los agentes juzgando su forma de recorrer el mapa, sin saber que en realidad están esquivando obstáculos que él no pueden ver, pero los mineros sí.

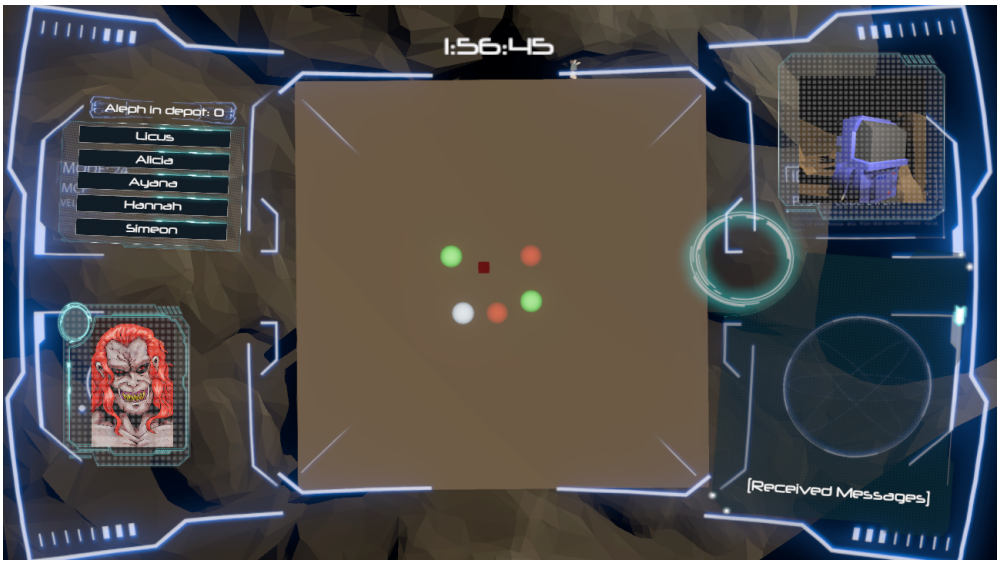

Figura 6.2: Imagen del prototipo con capa de niebla

Como en el juego final y en el prototipo, los personajes que el jugador puede manejar se encuentran en el cuadrante del centro. Además, para que el jugador se familiarizase lo antes posible con los controles y los mecanimos disponibles, se forzaría la aparición de un icono de Aleph situando una veta en el cuadrante central, de manera que los mineros exploradores marcaran la posición y generarán el icono interactivo lo antes posible.

A cada jugador se le explicó brevemente el objetivo de la prueba, los controles del juego y la información mostrada en la interfaz, y el cometido y funcionamiento de cada clase de minero. Se asistió en todo momento a los sujetos, recordándoles el uso de las mecánicas explicadas, y se realizaron los intentos necesarios hasta comprobar que cada jugador había aprendido las mecánicas del juego.

Para la evaluación de cada sujeto, primero se tomaría nota durante el desarrollo de la prueba de las veces que hubo que recordarle alguna mecánica, y del número de intentos hasta que éstas quedaran claras, y si en alguno de ellos se ha aproximado al número objetivo de Aleph. Después de ello se le preguntaría su opinión acerca de la claridad de la interfaz y la facilidad aprendizaje de los controles y mecánicas, y sobre el comportamiento de los agentes.

6.2. Construcción y configuración de la *build* del juego

Para exportar el proyecto y hacerlo ejecutable en los ordenadores del laboratorio reservado para la prueba, se creó una versión ejecutable de Unity para PC. Esta versión demandaría menos recursos a los equipos del laboratorio que la ejecución de la demo desde el propio editor de Unity

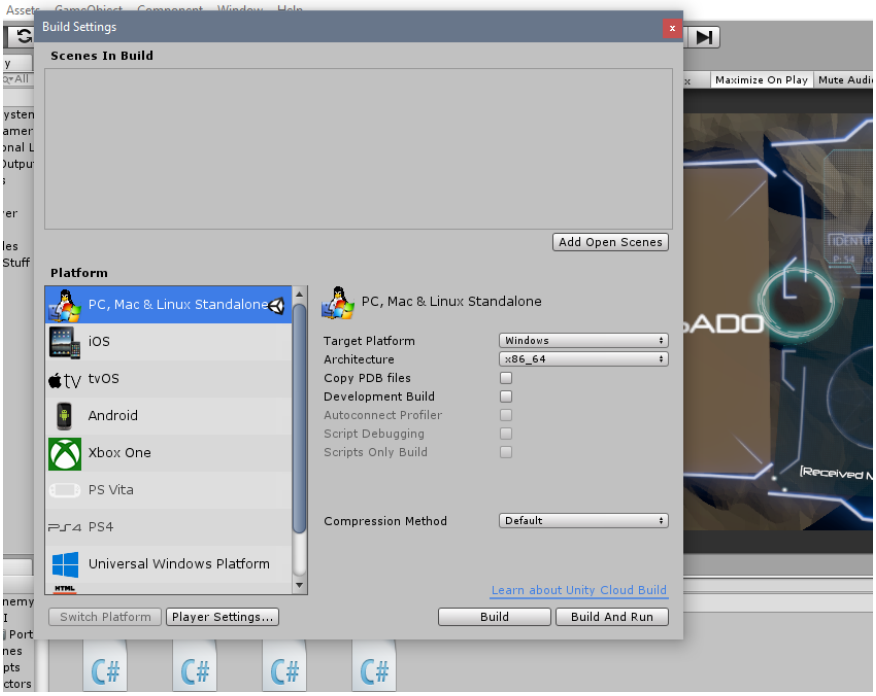

Figura 6.3: Imagen de la ventana para generar la versión ejecutable del prototipo

Para la ejecución del código Java y el sistema multiagente se hizo uso de una versión de jEdit ofrecida por Jason en su versión 2.2. De esta manera no habría que instalar el *plugin* de Jason en ningún entorno de desarrollo para la ejecución del código, realizando únicamente la ejecución del archivo *alephminers.mas2j* desde jEdit.

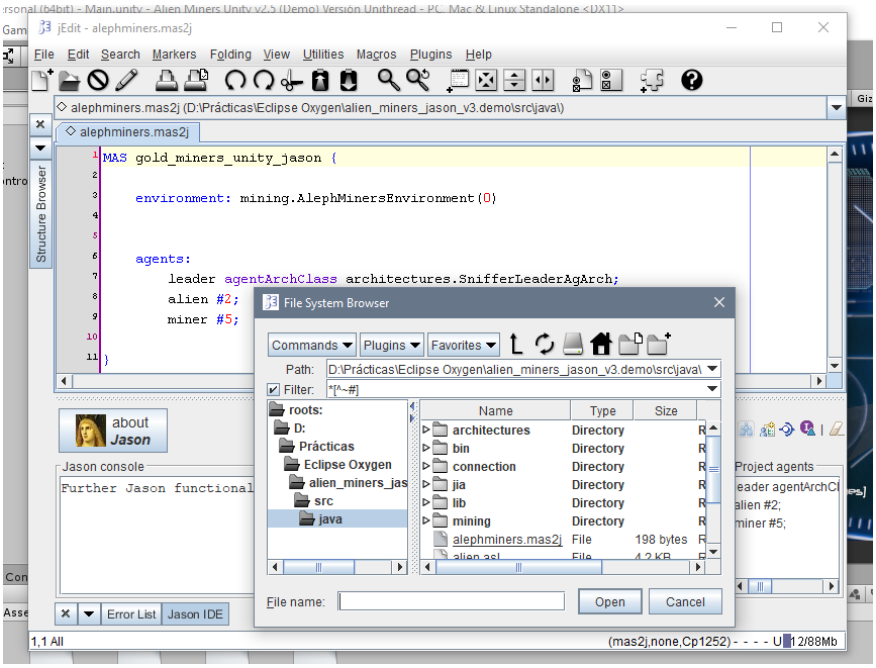

Figura 6.4: Imagen de jEdit para ejecutar el sistema multiagente

6.3. Desarrollo de la prueba y resultados obtenidos

Durante los primeros momentos, en la preparación de la demostración para la prueba, se comprobó que el rendimiento de ésta en los equipos del laboratorio era mínimo. Se pasó de un desempeño en el equipo de desarrollo de 50 *frames* por segundo (FPS) a uno de 20 FPS en el equipo para pruebas del laboratorio. Fue gracias a la ayuda de un compañero que se dio con la raíz del problema: la recepción y lectura de mensajes en Unity a través del *socket* estaba bloqueando la ejecución del hilo encargado de ejecutar todos los métodos *Update* implementados en el código. La solución propuesta por él fue implementar una lectura de mensajes asíncrona. Todos los detalles acerca del problema y su solución se encuentran en el apartado de implementación de *UniJason*, en el Capítulo 5.

También se detectaron durante los primeros ensayos fallos en la lógica de los recolectores, ignorando las ordenes del jugador ya que creían que estaban ocupados recogiendo Aleph cuando no lo estaban. La solución fue suprimir la inferencia e inserción de una creencia, *committed-to-gold*, que hacía que no se produjese el fallo.

En total, se evaluó a un grupo de 4 usuarios. Todas las pruebas realizadas a los distintos sujetos, menos la realizada al compañero que se interesó en resolver el problema de rendimiento, constaron de dos intentos, usando el primero como toma de contacto con los mecanismos, y el segundo para intentar superar el nivel. Entre todos los jugadores, sólo uno consiguió lograr extraer las 5 menas de Aleph, y además lo hizo en su segundo intento.

El alumno que realizo el experimento encontró confuso el feedback visual asociado con control —esto es, como se presenta al jugador qué personaje tiene seleccionado—. También subrayó el acabado de la interfaz y lo vistoso de ésta, echando en falta una referencia a los colores asociados a cada clase en cada uno de los botones con los nombres de los personajes. Al preguntarle acerca del comportamiento de los agentes, mencionó que "le parecían tontos" porque se movían de forma extraña, sin seguir una linea recta en ocasiones. Tras este comentario, se le mostró el tablero real —obstáculos, enemigos, mineros...— desactivando la *Mist Layer*. En ese momento, comprendió la razón por la que los mineros se desplazaban por el mapa, y concluyó que a él le resultaría más intuitivo conocer al menos las casillas ocupadas por obstáculos.

El otro probador, aparte de esbozar la solución al problema de rendimiento, comentó una serie de cambios respecto al *feedback* visual. Comentó que el parpadeo de las luces de los agentes es demasiado lento, que se hace en función de los *frames*, aunque esta impresión pudo estar influida por el mencionado problema de rendimiento. Añadió a este respecto que el haría que la luz fuera fija, de color blanco, para el jugador seleccionado. También comentó que se podrían dibujar flechas, desde el personaje a su destino, al pasar el ratón sobre los sitios a los que se puede mover el personaje seleccionado.

El codirector de este trabajo hizo sus apreciaciones respecto a la presentación de la información en pantalla. En concreto, mencionó que sería interesante cambiar a amarillo el color de la luz de los mineros recolectores que transportan Aleph. Así el jugador sabe qué recolector está ocupado. También mencionó que sería interesante cambiar la textura de las *tiles* que representan las casillas, haciendo que se viera explícitamente que son casillas, de forma que el jugador pueda medir más facilmente la distancia entre dos puntos para optimizar el tiempo del que dispone, calculando con mayor precisión el tiempo invertido en cada trayecto que realizan los personajes.

Finalmente, el director replanteó la interfaz entera. Eliminaría el *log* haciendo que cualquier información susceptible de aparecer en él se materializara en forma de *pop-up* temporal con la información en forma de texto o icono en el centro de la pantalla, o en forma de audio. Limitaría el uso de la cámara de los personajes a su uso en eventos específicos, por ejemplo, cuando un recolector empieza a cargar con Aleph, cuando un explorador lo encuentra, o cuando cuando avista a un enemigo. También propuso una redistribución de la interfaz, de modo que la parte inferior de la pantalla, un tercio del alto de ésta, estaría ocupada por fichas con los retratos y nombres de cada personaje, junto con iconos que indican su estado actual: en qué cuadrante están, qué están haciendo en cada momento, etc. Al resaltar cada ficha, se dibujaría una linea que uniría a ésta con el personaje asociado en el tablero; y a la derecha de las fichas se situaría un botón para pausar la acción. Por último, conservaría el contador regresivo en el centro superior de la pantalla.

Ambos directores señalaron lo innecesario de tener botones con nombre, cuyo papel ahora podrían asumir las fichas, y se apuntó como acierto la selección de personajes a través del teclado. También se consideró una re-implementación de los colores, estableciendo colores iguales pero de distinta tonalidad entre personajes de una misma clase, en lugar del mismo color para todos sus miembros.

6.4. Discusión sobre los resultados

6.4.1. Valoración de los resultados

A grandes rasgos, los usuarios que participaron en las pruebas entendieron el funcionamiento del prototipo, aunque en su segundo intento. Esto hace pensar que en la versión final del juego sería interesante presentar las mecánicas de una en una, a través de un tutorial, para evitar la frustración del jugador al no entenderlas.

También se ha señalado un falta de feedback visual, sobre todo en cuanto a la selección y control de personajes se refiere. Habría que ofrecer indicaciones al jugador de lo que hace el personaje seleccionado, para que no sienta que pierde el control de la partida debido al carácter autónomo de los personajes. Tal como está implementado ahora, los personajes simplemente se mueven a las posiciones indicadas, dejando al jugador la tarea de recordar más adelante por qué cada personaje se está movimiento de la manera que lo hace.

En esa dirección irían las propuestas de marcar a los recolectores que cargan oro con un color distinto, o el uso de flechas para determinar la posición a la que se está moviendo un minero. A esto se le podría añadir una mejor señalización del personaje seleccionado, aumentando el radio de su luz, o como también se propuso en la interfaz revisada, una línea en pantalla que conecte al personaje en el mapa con su ficha en la interfaz.

Al margen de las pruebas realizadas con los participantes, también se ha probado el desempeño del prototipo después de solucionar el problema de rendimiento que se manifestó el mismo día de la prueba. De esta forma, el uso de CPU, únicamente por parte del ejecutable del prototipo a máxima calidad gráfica, es de alrededor del 30 %, y si se ejecutan también los agentes en Java, su uso se acerca al 40 %. Para observar estos datos, hay que tener en cuenta que sólo se ejecutan 7 agentes concurrentemente, y que con la implementación del código en Unity se ha pretendido ser lo más eficiente posible, evitando realizar cálculos innecesarios cada *frame*.

La lectura que se puede sacar de ello es que, si el prototipo tiene un apartado gráfico y técnico modesto, muy alejado de los de videojuegos comerciales; y cada agente, relativamente simple, solicitando y ejecutando acciones sobre el entorno supone aproximadamente un 1 % de la CPU del equipo de pruebas, se podría pensar que la implementación de un sistema multiagente en un videojuego profesional podría no ser tan atractiva debido al consumo de recursos.

Además de esto, la ausencia de herramientas profesionales que faciliten la programación de estos sistemas también supone una barrera en su uso en el desarrollo de videojuegos, tanto de corte independiente como profesional.

Sin embargo, no todo son desventajas, ya que aún así los sistemas multiagente permiten crear lógicas bastante desacopladas. Jason permite al programador preocuparse únicamente de las precondiciones que se deben satisfacer para la ejecución de cada plan, dejando al agente la decisión de ejecutar un plan u otro en función de un criterio que también se puede estipular.

6.4.2. Revisión del Estado de la Técnica

A continuación se recorrerá el estado de la técnica, comentando y comparando los videojuegos, herramientas y proyectos allí expuestos con los resultados y reflexiones fruto de éste trabajo.

6.4.2.1. Inteligencia Artificial en Videojuegos Profesionales

En este apartado se reflexiona sobre cómo se podría mejorar la IA de los videojuegos mencionados usando sistemas multiagente BDI, y si tendría sentido para cada caso concreto.

Como hemos visto en el Capítulo 2, algunas IA aplicadas a la gestion de equipos, como la de *Killzone 3* (Guerrilla Games, 2011) y la de *Final Fantasy XV* (Square Enix, 2015), son bastante similares a la propuesta de *GoldMiners* (Rafael H. Bordini, 2007). Establecen una jerarquía en la que una entidad superior gestiona a personajes como recursos, siendo estos propiamente inteligentes.

Como se vio con Killzone 3 (Guerrilla Games, 2011), su IA ya es un sistema multiagente, escrito en Lisp y traducido a C++. La gestión que realizan los distintos niveles de su jerarquía son mucho más complejos que los que puede llegar a desarrollar el jugador con los agentes en este proyecto. Su sistema está hecho expresamente para no lastrar la experiencia del jugador, con una visión más táctica, enfocada a ganar la partida. Tal vez una forma de aportar algo a su robusto sistema, sea añadiendo motivaciones paralelas a los *bots*, como querer ser quien más jugadores mate en la partida o el que más lugares captura, haciéndolos parecer humanos no sólo en su eficacia, sino en el resto de su comportamiento.

Por otro lado, *Final Fantasy XV* (Square Enix, 2015) hace uso de arboles de comportamiento y máquinas de estado para representar el comportamiento de sus personajes. Su sistema está enfocado casi por completo a la batalla, así que para este tipo de cometidos tan sencillos, tal vez no resulte interesante el uso de sistemas multiagente, debido a lo poco que se utilizan, y lo ampliamente extendidos que están los árboles de comportamiento. Tal vez se podría utilizar un sistema de comunicación entre agentes como el que ofrece Jason para poder prescindir del "Director de IA". De esta forma, la colaboración en combate entre los personajes podría parecer más orgánica, consiguiendo que los compañeros sólo puedan ayudar a otros que están en peligro si realmente saben que lo están, no porque una entidad superior se lo diga.

Para *Dragon Age Inquisition* (Bioware, 2014), se podría aplicar un sistema multiagente para coordinar las habilidades entre los distintos integrantes de un equipo. De esta manera, se podría evitar que dos personajes hicieran ataques muy potentes al mismo tiempo, cuando con sólo uno de ellos se derrotaría al objetivo, propiciando el ahorro de magia o energía. O incluso coordinar ataques, debilitando al objetivo uno, y dando el golpe de gracia el otro. Sin embargo, la dificultad que supondría implementar su sistema de evaluación de habilidades en código ASL, obligaría prácticamente al uso de otros sistemas que se acoplasen a Jason a través de sus acciones internas.

En *The Last of Us* (Naughty Dog, 2013), nuevamente, la aplicación de sistemas multiagente para modelar el comportamiento de los compañeros no tiene ningún sentido, ya que existen sistemas mucho más simples y eficaces. Un sistema multiagente sería desproporcionado para implementar la IA de un sólo compañero, aunque tal vez sí tendría sentido para hacerlo con la de los enemigos, de forma similar a como se hizo con los *bots* de Killzone 3 (Guerrilla Games, 2011). En cambio, si el cometido de la IA fuera emular el comportamiento humano, tal vez se podría condicionar el funcionamiento del sistema de exploración de *The Last of Us* (Naughty Dog, 2013) con estados emocionales o personalidades, como se hace en este proyecto.

6.4.2.2. Herramientas de IA para Unity

En esta sección se considera si habría sido útil hacer uso de otras herramientas para llevar a cabo el presente proyecto, y compararlas con lo que ofrece Jason y, con ello, UniJason.

El uso de redes neuronales, como las que proporciona *ANN Perceptron* no tendría mucho sentido en el proyecto. Sus aplicaciones están más dirigidas al aprendizaje reforzado, y sería muy difícil modelar un videojuego en el que la IA de sus personajes se apoyara en redes neuronales, puesto que primero habría que entrenarlas con comportamientos de jugadores. Y tampoco tiene sentido porque para modelar comportamientos similares a los humanos hay formas mucho más directas de hacerlo: Sistemas multiagente, árboles de comportamiento, lógica difusa, etc.

Las redes bayesianas por sí sólas tampoco constituyen una IA. Se tienen que utilizar dentro de otras estructuras para modelar comportamientos complejos. Aplicarlas, por ejemplo, a árboles de comportamiento sería interesante, ya que se tendría la secuencialidad de los árboles por un lado, y la lógica probabilistica de las redes bayesianas, para la toma de decisiones, por otro.

Los árboles de comportamiento, sin embargo, son un método muy extendido para modelar la IA de los NPCs. Es poco costoso computacionalmente, su representación es explícita y visual, y permite crear secuencias de comportamientos complejos. Sin embargo, existe cierto acoplamiento entre los nodos de un árbol (generalmente a través de la estructura de pizarra), creando fuertes dependencias entre nodos.

En cambio, Jason permite crear conjuntos de planes que se ejecutan bajo circunstancias concretas, independientemente de los eventos que hayan generado esas circunstancias. Es decir, permite modelar cómo se comporta un NPC en función del contexto, pudiendo éste decidir qué acción realizar —si hay varias que se pueden ejecutar bajo un mismo contexto—, siguiendo un criterio de selección de planes (Round-Robin, por prioridad explícita...) que también puede ser especificado por el programador.

Además de ésto, permite la comunicación directa entre agentes. Esto permite de forma sencilla compartir información acerca del entorno, que puede ser verdadera o falsa, dependiendo de los planes que produzcan dicha comunicación.

El principal inconveniente de Jason, eso sí, es su escasa aplicabilidad. Apenas hay herramientas que faciliten la implementación de sistemas multiagente, por no hablar de la escasa documentación que hay al respecto. También es problemática la programación en sí misma, teniendo que estructurar muy bien los planes, que están programados en un lenguaje declarativo relativamente desconocido. Es fundamental saber qué efectos produce cada plan y en función de cuáles se ejecuta, para evitar comportamientos imprevisibles, y a veces indeseados.

Sin embargo, si el objetivo es precisamente lograr una cierta imprevisibilidad de cara al usuario, Jason es una herramienta que se presta a ello. De hecho, se pueden observar ciertas similitudes entre la forma en que los diseñadores de videojuegos que implementan múltiples mecánicas que chocan entre sí para favorecer la jugabilidad emergente; y la forma en que se implementan los planes en Jason.

Aunque también cabe decir que hacer personajes excesivamente impredecibles podría conducir a una mala experiencia de usuario. Ya se probó con algunos jugadores para el videojuego *The Last Guardian* (genDESIGN, SCE Japan Studio, 2016), en el que el jugador acompaña a una especie de quimera que se comporta de forma muy orgánica, desobedeciendo en ocasiones las órdenes del jugador en detrimento de aparentar tener intereses propios.

Lamentablemente, Jason y *UniJason* por si solos no tienen nada que hacer en cuanto a recursos contra *suites* de IA como *Emerald* y *Apex AI Utility*, que ponen a disposición del usuario múltiples herramientas de autoría (visuales, incluso) que se utilizan en conjunto para dar forma a un complejo sistema de IA.

El primero no tiene tanto potencial, pues está orientado a que el desarrollador utilice características predefinidas claramente enfocadas al desarrollo de videojuegos RPG, aunque evitándole tener que preocuparse de programar. Los comportamientos y temperamentos que ofrecen a los desarrolladores están predefinidos, de manera que no se pueden definir nuevos arquetipos o personalidades, como sí permite Jason. Apex AI Utility en cambio es algo más sofisticada, general y polivalente, ofreciendo árboles de decision y funciones de evaluación a través de lógica difusa y curvas de decisión. Ambas disponen de recursos de aprendizaje, como tutoriales, documentación y APIs, y un soporte continuado.

Y por supuesto, *UniJason* supone una gran mejora respecto al trabajo desarrollado en *Isomonks* (Sánchez-López y cols., 2016). Se constituye como una herramienta genérica y extensible para cualquier tipo de videojuego desarrollado en Unity, y no sólo para una demo sencilla realizada sobre el plugin *IsoUnity* (Pérez-Colado y Pérez-Colado, 2014).

6.4.2.3. Proyectos e investigaciones de IA para videojuegos

Por último, en este apartado se discuten algunas de los estudios realizados en los proyectos mencionados en el estado de la técnica, poniéndolos en relación con este TFG, y viendo cómo refuerza sus planteamientos u ofrece alternativas a sus objetivos. Lógicamente, se ignora el apartado de *GoldMiners*, puesto que este TFG ya constituye una mejora sobre dicho trabajo.

En primer lugar, el trabajo de Boyang Li y Mark O. Riedl (Li y Riedl, 2010). En él se persigue mejorar la experiencia del jugador y favorecer la rejugabilidad haciendo partícipe al jugador de la coherencia narrativa de las misiones dentro de un videojuego. Recordar que su planteamiento se resumen en poder adaptar los objetivos y recompensas de una misión o secuencia de misiones a los intereses del jugador dentro del juego.

Su trabajo propone un algoritmo concreto para llevar esto a cabo, y no sería difícil, en términos teóricos, enmarcar el uso de sistemas multiagente de en estos términos. Además, algo que no se trata en el texto es cómo extrae esa información del jugador, y Jason podría servir para unificar ambos frentes.

Se podría diseñar un agente que monitorice la actividad del jugador en el juego, observando sus decisiones, qué tipos de misiones elige hacer antes que otras, de qué forma interactúa en los diálogos... formalizando una base de planes que se ejecutarían según la actividad del jugador, generando creencias sobre él.

Estas creencias podrían ser utilizadas por el mismo monitor para su uso en planes preconcebidos por los desarrolladores, de manera que al jugador que emplea poco tiempo en las conversaciones se le ofrezcan objetivos más enfocados a la acción, por ejemplo.

También cabe destacar que en este TFG se ha querido explorar el uso de la incertidumbre en el comportamiento de los personajes, confiriéndoles una personalidad, que el jugador desconoce a priori, y que puede sesgar su comportamiento en mayor o menor medida. Esto serviría como medio para favorecer la rejugabilidad y mejorar la experiencia de juego, al igual que pretende el sistema de Li y Riedl. Tal vez combinar la adaptabilidad del sistema de misiones con el desarrollo de personajes que representen personalidades o arquetipos reconocibles (como comentan Ocio y López Burgos en su artículo (Ocio y Burgos, 2009)) y comportamientos ligeramente impredecibles sirva para mejorar aún más la experiencia de usuario.

Respecto al trabajo de Ocio y López Burgos, lo que más interés tiene son sus comentarios acerca del rendimiento de los sistemas multiagente en videojuegos sandbox. En resumen, proponen una jerarquía de capas que "desconectan" el comportamiento plenamente funcional de los agentes del sistema cuando estos comportamientos, o sus efectos, no son directamente apreciables por el jugador. De esta forma, se ahorra esfuerzo computacional que puede ser destinado a otras tareas más relacionadas con el computo gráfico del videojuego, por ejemplo.

Bajo los modestos y poco precisos resultados en términos de rendimiento del presente trabajo, parece sugerirse que la implementación de sistemas multiagente, basados en Jason, se presume costosa para videojuegos de mundo abierto. Es entonces donde la arquitectura LOD de Ocio y López Burgos jugaría un papel fundamental a la hora de llevar a cabo dicha implementación.

Proponen que para los agentes de baja prioridad no se actualicen al ratio máximo permitido por el sistema, y a esta solución se le podría añadir también el uso de árboles de decisión o máquinas de estados para los agentes más básicos y/o distantes al jugador, reduciendo los costes significativamente.

Por último *Narrative AI* se postula como un modelo BDI en el que, fundamentalmente, la clasificación de sus operadores se realiza en el dominio de las emociones. De esta forma, el aspecto más interesante de su modelo es cómo enajena el tratamiento de las emociones de los personajes, haciéndolas incontrolables por ellos mismos aunque estén definidas por su personalidad. Aquel que controla cómo afectan los cambios e interacciones con el entorno a las emociones de los personajes es el planificador.

Este modelo podría implementarse con Jason, utilizando un agente como planificador de esas emociones. Sería necesario implementar un agente capaz de percibir todos los cambios del entorno y de escuchar, utilizando una versión adaptada de la arquitectura *SnifferLeaderAgArch* desarrollada para este TFG, para la escucha de todos los mensajes dichos entre los personajes.

La modificación de la base de creencias de los personajes por parte del planificador se podría realizar infiriendo los cambios en dicha base teniendo en cuenta los rasgos emocionales del personaje, para después enviárselos. De esta manera, el agente no almacena planes que definan cómo debe comportarse ante cambios emocionales.

Capítulo 7

Conclusiones

Tal como se empezó contando en la Introducción, este trabajo se enmarca en un contexto en el que el desarrollo de IA para mejorar la jugabilidad no experimenta su mejor momento. Ante esa realidad, este trabajo pretendía enseñar las posibilidades de los sistemas multiagente en videojuegos. De este modo, se propuso su uso en videojuegos desarrollados en Unity.

Para lograrlo, primero ha sido necesario construir un conector entre Unity y Jason, *UniJason*, para comunicar la mente de los agentes en Jason con su cuerpo en Unity.

Después, para el diseño del videojuego propiamente dicho, se contó con la ayuda del diseñador de videojuegos Javier Vela Ramos para encontrar una forma de lucir el sistema BDI. Posteriormente, se desarrolló y finalmente se probó, constatando el valor de la idea y el diseño del videojuego.

En la presente memoria se han revisado otros trabajos que ahondan en el desarrollo de inteligencias artificiales de coordinación de agentes, y en la mejora de la narrativa a través de la imitación de comportamientos humanos o la adaptación de la narrativa a los objetivos del jugador.

También se han documentado las pautas seguidas en el proceso de desarrollo, así como el desarrollo en sí mismo, dividiendo ambas partes del trabajo en versiones. Además, se han expuesto las pruebas realizadas y los resultados obtenidos en tales pruebas. A éstos, se les suman otros observados por el autor de este trabajo final, después de la implementación del sistema de personalidades que lo concluye.

En las siguientes páginas se resumen los resultados del presente proyecto y las lineas de trabajo futuro a las que da lugar.

7.1. Resumen de resultados

Estos son los tres grandes objetivos que se propusieron en el Capítulo 3 son:

- Comprender y valorar lo que puede ofrecer la aplicación de *Jason* a videojuegos. Se tratará de responder a la pregunta sobre la utilidad de aplicar este tipo de sistemas multiagente frente a otros sistemas.
- Desarrollar un conector que una el sistema multiagente (*Jason*), con el entorno de desarrollo de videojuegos *Unity*, y así comunicar la *mente* y el *cuerpo* de cada personaje, implementados en respectivas herramientas. Posteriormente, y sobre este conector, diseñar y prototipar un videojuego que sirva para ilustrar el potencial del paradigma BDI.
- Probar dicho juego con usuarios reales para asegurar la usabilidad del mismo y la satisfacción del usuario, además de validar si la inclusión de comportamientos inteligentes es relevante desde el punto de vista de la experiencia del jugador.

Respecto al primer objetivo, como se comenta a lo largo de la discusión, se concluye que efectivamente los sistemas multiagente, en concreto Jason, pueden ayudar a mejorar los sistemas ya existentes en videojuegos comerciales. También se defiende su utilidad frente a otras aplicaciones de IA en videojuegos, y se apoya en otros trabajos para ampliar aún más la utilidad de dichos sistemas.

Para el segundo, se logró diseñar con éxito el controlador genérico y extensible, y para mostrar su funcionamiento se adaptó la demo *GoldMiners* de Jason en Unity. Sobre ella se diseñó el prototipo del videojuego, *Miners of the Broken Planet*. A partir de él, se desarrollaron sus gráficos, mecánicas, controles e interfaz; y los sistemas de clases y de personalidades, que ilustran el potencial del paradigma BDI.

Respecto al último de los objetivos, se probó el juego con usuarios, constatando su valor como videojuego, y recibiendo una gran cantidad de *feedback* que será tenido en cuenta más adelante. Sin embargo, dada la ambición y envergadura del proyecto, no se han podido realizar todas las pruebas con usuarios que habrían sido deseables. Así, ha quedado por probar el sistema de personalidades que no obstante sí ha sido implementado en fase alpha, y ha sido probado con resultados satisfactorios.

Entre los aspectos positivos de este desarrollo destaca la oportunidad que ha supuesto para profundizar en aspectos inexplorados a lo largo de los estudios de grado, como es el uso de sistemas multiagente basados en arquitecturas cognitivas como la que ofrece Jason, o el uso de entornos de desarrollo de videojuegos, como es el caso de Unity. También se ha aprendido cómo hacer documentos de diseño de videojuego, y profundizado en C# y el uso de *sockets* UDP.

Además se han aplicado conocimientos de toda la carrera. Desde lo visto sobre programación orientada a objetos en Tecnología de la Programación, pasando por la sincronización de procesos y la gestión de mensajes simultáneos de Programación Concurrente, hasta llegar al uso de AgentSpeak, con similitudes con lenguajes declarativos como Prolog, visto en Programación Declarativa.

Además, este proyecto es fruto de la unión de los dos frentes de la Ingeniería Informática qué más interesan a su autor: la Inteligencia Artificial y los Videojuegos.

Sin embargo, el desarrollo del trabajo final no ha estado exento de dificultades, ocasionadas en su mayoría por la inexperiencia con las dos herramientas principales utilizadas, y por la escasa comunidad que desarrolla y realiza aportaciones sobre Jason.

7.2. Trabajo futuro

Para concluir, se comentan a continuación las líneas de trabajo futuro que se abren a partir de este proyecto, en los frentes de *UniJason* y de *Miners of the Broken Planet*.

Respecto a *UniJason*, podría mejorarse la comunicación utilizando el protocolo TCP en lugar del UDP. Es cierto que el proyecto, con sus pruebas, está pensado para ejecutarse sobre un mismo equipo, evitando lidiar con los múltiples problemas que se pueden dar en Internet, pero sería interesante modificar el protocolo precisamente para posibilitar su uso a través de Internet. De esta forma se podría ejecutar la IA de los agentes y el modelo del juego en equipos remotos.

Se podría llevar Jason a C# y prescindir de *UniJason*. Facilitaría la implementación de sistemas multiagente en juegos desarrollados en Unity, haciéndola mucho más directa y sencilla, evitando también el retardo producido por el paso de mensajes y dando lugar a un contacto directo entre los agentes y su entorno.

En el Capítulo 6 también se ha hablado de la inexistencia herramientas para agilizar la escritura de sistemas multiagentes en Jason. A este respecto, se está hablando ya del desarrollo de una herramienta que permita editar y depurar desde Unity el código ASL de los agentes, facilitando y agilizando su implementación.

Sin embargo, *Miners of the Broken Planet* al ser la parte más grande del proyecto, es también la que más margen de mejora ofrece. Por un lado, se debería probar el sistema con más usuarios y habiendo incorporado ya todas las mejoras debidas al feedback recibido durante la primera sesión de prueba. Por otro, se puede ampliar lo que a día de hoy es un simple prototipo, creando más niveles laberínticos, dificultando las tareas de los mineros en cada partida y creando una sensación de progresión en el jugador.

En fases tempranas de su desarrollo, se consideró que añadir una capa narrativa por encima sería muy interesante. Se puso como ejemplo *Papers, please* (Lucas Pope, 2013), un videojuego de producción independiente que crea un conflicto de intereses entre lo que se supone que debe hacer el jugador y lo que tiene que hacer para sobrevivir.

La inteligencia implementada para los mineros también alberga mucho margen de mejora. Podrían añadirse más personalidades o arquetipos a las ya existentes para aumentar las posibles combinaciones que pueden darse en un equipo. De esta forma, se favorecería la rejugabilidad, y se evitaría que el jugador aprendiese una forma determinada de romper la jugabilidad, puesto que ésta estaría aún más condicionada por la incertidumbre en la forma de ser de los personajes.

También se podrían implementar distintos grados de confianza en función de lo que unos agentes saben de otros. Así, si alguien sabe que un agente ha dado una información errónea, puede hacer que su confianza en la información que ofrece ese agente disminuya. Incluso dicha disminución podría estar condicionada por las consecuencias de usar dicha información errónea; por ejemplo, disminuyendo un poco si sólo le ha hecho perder el tiempo, o bastante más si ha conducido a la muerte de un miembro del equipo.

Adicionalmente, se podrían agregar lazos entre los personajes, de modo que, si un agente sabe que otro con quien tiene una relación estrecha está en peligro, podría ignorar órdenes del jugador con tal de salvarlo. Uniéndolo con lo anterior, estos lazos podrían condicionar la confianza que se tiene en otros agentes, y cómo varía.

A pesar de que todos estos cambios serían interesantes, y harían del sistema uno mucho más inteligente, los desarrolladores de estas posibles mejoras deben cuidarse de no perder la perspectiva de videojuego. Como ya se comenta en la discusión, hacer personajes excesivamente impredecibles puede conducir a una mala experiencia de juego.

En cualquier caso, ya se ha hablado con Mercury Steam, la empresa más importante de videojuegos en España, para presentarles una versión refinada del prototipo ya hecho. La idea es renovar la interfaz, siguiendo las propuestas del Capítulo 6, e incluir música adaptativa a niveles de distinta duración, aparte de gráficos más acorde a la obra a la que pretende rendir tributo, *Raiders of the Broken Planet* (Mercury Steam, 2017).

Referencias

Apex Game Tools. (2016). *Apex AI Utility.*

Bethesda Softworks. (2006). *The Elder Scrolls: Oblivion.* [Multiplataforma].

Bethesda Softworks. (2011). *The Elder Scrolls: Skyrim.* [Multiplataforma].

Bethesda Softworks. (2015). *Fallout 4.* [PS4, Xbox One, PC].

Bioware. (2014). *Dragon Age Inquisition.* [Multiplataforma].

Black Horizon Studios. (2015). *Emerald.*

Crema Games. (2017). *Immortal Redneck.* [Multiplataforma].

genDESIGN, SCE Japan Studio. (2016). *The Last Guardian.* [PS4].

Guerrilla Games. (2011). *Killzone 3.* [PS3].

Haven Made, Craigz. (2017). *Warcube.* [PC].

Jacky Chen. (2016). *Discrete Bayesian Network.*

Jonathan Nolan. (2016). *Westworld.* [HBO Streaming].

Li, B., y Riedl, M. O. (2010). *Planning for Individualized Experiences with Quest-Centric Game Adaptation.*

Lucas Pope. (2013). *Papers, please.* [PC].

Mercury Steam. (2017). *Raiders of the Broken Planet.* [PS4, Xbox One, PC].

Naughty Dog. (2013). *The Last of Us.* [PS3, PS4].

Ocio, S., y Burgos, J. A. L. (2009). *Multi-agent Systems and Sandbox Games.*

Opera Soft. (1987). *La abadía del crimen.* [Multiplataforma].

Opsive. (2014). *Behavior Designer.*

Outsider Studios. (TBD). *Project Wight.* [PC].

Peinado, F., Cavazza, M., y Pizzi, D. (2008). *Revisiting Character-Based Affective Storytelling under a Narrative BDI Framework.*

Pérez-Colado, I. J., y Pérez-Colado, V. M. (2014). *Un conjunto de herramientas para unity orientado al desarrollo de videojuegos de acción-aventura y estilo retro con gráficos isométricos 3d.*

Rafael H. Bordini, M. W., Jomi Fred Hübner. (2007). *Programming Multi-agent systems in AgentSpeak using Jason.* Wiley.

Rockstar Games. (1997 - 2015). *Grand Theft Auto.* [Multiplataforma].

Square Enix. (2015). *Final Fantasy XV.* [PS4, Xbox One, PC].

Sánchez-López, A., Romero, F., y Martín-Solís, H. (2016). *Un Sistema de Control Autónomo para Personajes de Videojuegos Basado en el Modelo Cognitivo Creencia-Deseo-Intención.*

Virtualstar. (2018). *ANN Perceptron.*

Apéndice A

Introduction

The applications of Artificial Intelligence (AI) to Game Development is a topic of great interest to experts as well as to consumers. In the previous years, this synergy has helped the appearance of new tools to attempt to reduce the number of resourecs and the great amount of effort needed for any great production.

Meanwhile, the technological progress and the consumers' improved access to more advanced hardware for more affordable prices has easen the development of videogames that offer greater worlds to explore, inhabitated by beings that interact both with the player and with their environment.

It has also been made more affordable, from the perspective of businesses, to compete in the videogame industry, offering functionalities and technological breakthroughs to levels never seen before. This makes the developers to create new ways to stand out in a medium with a growingly demanding public, either to make a way for themselves in an independent market saturated with games of all the kinds known to men, or to keep the lead position in the case of greater companies, renovating the way they create the new installments of their successful franchises.

Nevertheless, in spite of these advances, the Artificial Intelligence applied to gameplay hasn't improved much in recent years. Videogames such as Fallout 4 (Bethesda Softworks, 2015) display some shortcomings present in previous titles, such as TES: Skyrim (Bethesda Softworks, 2011), —which are inherited from the previous title TES: Oblivion (Bethesda Softworks, 2006)— in the field of the AI of NPCs (Non playable characters). Such stillness manifests even after the passing of almost a decade.

This stagnation may be due to several related factors. The first of these factors is a matter of budget; the second, to the great difficulty that is present when attempting to advertise the improvements provided by the Artificial Intelligence in a videogame instead of graphics that speak for themselves; and lastly, the subtle that the results are, even for an effort that involves many resources.

Apart from AI, the interest in emergent gameplay is having a greater role in the recent years. This is defined as the set of non prefabricated dynamics that are created in response to the interactions of the player inside the game. That is, the uses that the player can give to the different tools that are contained in the game in order to resolve problems in ways that were not premeditated explicitly in the creation of the videogame.

In the search of a way to grant sense to the use of advanced AI in the development of a videogame, in an affordable way that is also relatively simple, the most popular choice is to use a paradigm that represents the behavior of the characters of a videogame in a non scripted way. One of the other objectives are to avoid complexity and the coupling between the tasks that can be found in structures such as behavior trees (Behavior trees of Unreal Engine). That is why that the way to represent the knowledge of agents, the BDI (Beliefs, Desires and Intentions) cognitive model is applied. This is characterized by the proposal that states that the mental models that human beings have in relation to our environment are theories about it. These theories are represented by Beliefs, that can be either true or false, that offer a context to any possible plan to develop, which are Intentions, that assume the ways to an end, and so, the Desires of the agent.

A.1. Purpose of the work

Taking into account these premises, this Final Assignment is developed, whose purpose is to explore and justify the use of new approaches for the application of Artificial Intelligence in videogames. More specifically, applying the BDI cognitive architecture in the behavior of non playable characters. Also, as all Final Assigments, it has as objective the application of many of the knowledge adquire along the degree, deepening into fields and new tools.

To reach these intentions, two main tools in their fields have been used: the Java based interpreter, Jason[1] for the development of the AI of the characters from the videogame, supported by a Multiagent System; and Unity[2], the reference videogame development environment in the industry.

This project is made of two differenciated parts:

The first one consists in the development of a generic, extensible tool that links the videogame, implemented using the Unity engine, with the artificial intelligence of the characters, developed in Java using Jason.

The second part consists in the design and development of a videogame that makes use of the technology offered by Jason, with characters that act in different ways according to what they know about themselves and the environment. Taking into account the complexity of this task, the support of the codirector of the assignment was requested: Javier Vela Ramos, who works as Game Designer at Mercury Steam. With his help, a design was elaborated that displays some of the great characteristics that BDI offers to the videogame.

[1]Jason Interpreter: http://jason.sourceforge.net/
[2]Unity Engine: https://unity3d.com

A.2. Assignment structure

Having explained the purpose of this assignment, and the context where it develops, the content of the following chapters present in this document is described below. Chapter 2 refers to the state of the art, which revises, in an orderly manner, the AI applications in some concrete videogames in a theoretical way; real tools, that already exist for Unity, used to develop AI characteristics; and lastly, other projects and insights similar to this. Chapter 3 details the objectives for this project, specifying the requirements used as foundation for all the development ahead. The general work schedule, the methodology and the tools used in the communication and organization of the work team that have been used are presented in Chapter 4.

Chapter 5 exposes the bulk of the structural activities of the software development of this work. In it, analysis, design and implementation of the generic linker between Jason and Unity, called *UniJason*, are detailed. Details about the videogame prototype, called *Miners of the Broken Planet*, are presented in a formalised way as a videogame design document.

The trials performed on the videogame prototype and the obtained results are discussed in the Chapter 6. Lastly, Chapter 7 collects the conclusions about this project, which have generated various interesting lines of future work to follow up the improvement of both the linker as well as the developed videogame prototype.

Apéndice B

Conclusions

As started in the Introduction, this work is framed in a context in which the development of AI to improve gameplay is not experiencing its best moment. Faced with this reality, this work aimed to show the possibilities of multi-agent systems in video games. Thus, it was proposed for use in video games developed in Unity.

To achieve this, it has first been necessary to build a connector between Unity and Jason, UniJason, to communicate the mind of the agents in Jason with his body in Unity.

After that, for the design of the videogame itself, the videogame designer Javier Vela Ramos helped to find a way to show off the BDI system. Later, it was developed and finally tested, verifying the value of the idea and the design of the videogame.

In this report we have reviewed other works that delve into the development of intelligences artificial for the coordination of agents, and the improvement of the narrative through the imitation of human behavior or the adaptation of the narrative to the objectives of the player.

The guidelines followed in the development process have also been documented, as well as the development itself, dividing both parts of the work into versions. In addition, the tests performed and the results obtained in these tests have been presented. To these, there are others observed by the author of this paper, after the implementation of the system of personalities that concludes it. The following pages summarize the results of the present project and the lines of future work to which it gives rise.

B.1. Summary of results

These are the three main objectives that were proposed in Chapter 3 are:

- Understand and appreciate what Jason's application to video games can offer. We will try to answer the question about the usefulness of applying this type of multi-agent system compared to other systems.
- Develop a connector that connects a multi-agent system (Jason), with the Unity videogame development environment, and thus communicate the mind and body of each character, implemented in respective tools. Then, on this connector, design and prototype a videogame to illustrate the potential of the BDI paradigm.
- Try this game with real users to ensure the usability of the game and user satisfaction, as well as to validate whether the inclusion of intelligent behaviors is relevant from the point of view of the player's experience.

With respect to the first objective, as discussed throughout the discussion, it is concluded that multi-agent systems, specifically Jason, can effectively help to improve existing systems in commercial video games. It also defends its usefulness compared to other AI applications in videogames, and it is supported by the other works to further expand the usefulness of such systems.

For the second, the generic and extensible controller was successfully designed, and to show how it works, Jason's GoldMiners demo was adapted in Unity. The prototype of the videogame, Miners of the Broken Planet, was designed around it. From it, its graphics, mechanics, controls and interface; and class and personality systems, which illustrate the potential of the BDI paradigm, were developed.

About the last of the objectives, the game was tested with users, verifying its value as a videogame, and receiving a great deal of feedback that will be taken into account later on. However, given the ambition and scale of the project, it has not been possible to carry out all the tests with users who would have been desirable. Thus, the system of personalities has yet to be tested, but it has been implemented in the alpha phase, and has been tested with satisfactory results.

Among the positive aspects of this development is the opportunity it has provided to delve into unexplored aspects throughout the degree studies, such as the use of multi-agent systems based on cognitive architectures such as the one offered by Jason, or the use of video game development environments, such as Unity. He has also learned how to make video game design documents, and delved into C# and the use of UDP sockets.

In addition, knowledge from the whole career has been applied. From the view on object-oriented programming in Programming Technology, through the synchronization of processes and the management of simultaneous messages of Concurrent Programming, to the use of AgentSpeak, with similarities with declarative languages such as Prolog, seen in Declarative Programming.

Also, this project is the result of the union of the two fronts of Computer Engineering which most interest its author: Intelligence Artificial and Video Games.

However, the development of the work has not been exempt from difficulties, caused mostly by inexperience with the two main tools used, and by the scarce community that develops and contributes to Jason.

B.2. Future work

To conclude, the future lines of work that will be opened up from this project, on the fronts of UniJason and Miners of the Broken Planet, are discussed below.

About to UniJason, communication could be improved by using TCP instead of UDP. It is true that the project, with its tests, is designed to run on the same computer, avoiding the multiple problems that can occur on the Internet, but it would be interesting modify the protocol precisely to enable its use over the Internet. This way you could run the AI of the agents and the game model on remote computers.

Jason could be taken to C# and we could do without *UniJason*. It would facilitate the implementation of multi-agent systems in games developed in Unity, making it much more direct and simple, also avoiding the delay caused by the passage of messages and giving rise to direct contact between agents and their environment.

Chapter 6 has also discussed the lack of tools to speed up the writing of multi-agent systems in Jason. In this respect, we are already talking about the development of a tool that allows to edit and debug the ASL code of the agents from Unity, facilitating and speeding up its implementation.

However, *Miners of the Broken Planet*, being the largest part of the project, also offers the most room for improvement. On the one hand, the system should be tested with more users and all the improvements due to the feedback received during the first test session should have been incorporated. On the other hand, you can expand what is now a simple prototype, creating more labyrinthine levels, hindering the tasks of the miners in each game and creating a sense of progression in the player.

In the early stages of its development, it was considered that adding a narrative layer above it would be very interesting. An example was *Papers, please* (Lucas Pope, 2013), an independently produced video game that creates a conflict of interests between what the player is supposed to do and what he has to do to survive.

The intelligence implemented for the miners has also room for improvement. More personalities or archetypes could be added to the existing ones to increase the possible combinations that can occur in a team. In this way, it would favour replayability, and would prevent the player from learning a certain way of breaking the gameplay, since this would be even more conditioned by the uncertainty in the characters' way of being.

Different degrees of reliance could also be implemented depending on what some agents know about others. Thus, if someone knows that an agent has given the wrong information, they can make their reliance in the information offered by that agent decrease. Even such a decrease could be conditioned by the consequences of your misinformation, for example, decreasing somewhat if it has only wasted your time, or significantly more if it has led to the death of a team member. In addition, ties could be added between the characters, so that if an agent knows that another agent with whom he has a close relationship is in danger, he could ignore the player's orders to save him. Combined with the above, these links could condition the reliance that is held in other agents, and how it varies.

While all of these changes would be interesting, and make the system much smarter, the developers of these possible improvements must be careful not to lose sight of the game perspective. As discussed in the discussion, making overly unpredictable characters can lead to a bad gaming experience.

In any case, we have already spoken to Mercury Steam, the most important videogame company in Spain, to present a refined version of the prototype already made. The idea is to renew the interface, following the proposals in Chapter 6, and to include adaptive music at different levels of duration, apart from graphics more in line with the work to which he intends to pay tribute, *Raiders of the Broken Planet* (Mercury Steam, 2017).